Statistics and Runtime Verification

Andrew Gauci

Faculty of ICT

University of Malta

Submitted for the degree of Bachelor of Science

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Statistics And Runtime Verification

submitted is my work, except where acknowledged and referenced.

Andrew Gauci

June 09

Acknowledgements

First of all, I would like to thank Gordon J. Pace for his supervision and support throughout my thesis. Not only was he of constant help throughout the year, but also his ideas and discussions often led to interesting developments within my presented work.

Thanks also go to Christian Colombo, whose tool LARVA was essential for the development of my thesis. Moreover, his insight and constant patience shown through frequent alterations made by him to the tool as required by my work was invaluable.

Abstract

Software is influencing our lives in more frequent and crucial ways. This implies an ever increasing need for reliable software, whereby such reliability can be guaranteed through a method which guarantees software correctness i.e. that the system implementation adheres to its corresponding specification. This field is commonly known as software verification, and admits multiple techniques such as testing and model checking. However, given the intractability of verifying each execution trace, and the lack of coverage implied by testing we require a new approach to the field. This is provided by runtime verification, whose concern is the verification of the currently executing system trace, and can be considered as an extension to testing with augmented guarantees through the use of more powerful specification languages.

We theorise that augmenting runtime verification with statistics is advantageous. Firstly, it is often better to watch for indicators of failure than to wait for failure. Also, statistics allow for the straightforward specification of non-functional requirements, such as specifications related to performance and reliability (as opposed to most runtime verification logics which solely focus on functional requirement specification). Moreover, from a utilitarian point of view the collection of statistics over runtime executions allows for applications to fields such as performance profiling, intrusion detection, user modeling and quality of service. Finally, augmenting a logic used for runtime verification with statistical capabilities may increase the logic expressivity, or at least increases the conciseness and intuitive expression of certain system specifications.

The augmentation of runtime verification with the notion of statistics implies the need for an underlying runtime verification framework. We identify this framework to be LARVA, an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs. Based on Dynamic Automata with Timers and Events (DATEs) as the underlying mathematical framework, LARVA allows for the expression of (i) Temporal aspects dealing with consequentiality and real time (ii) contextual aspects with the possibility of monitoring objects either globally, grouped according to their context or individually (iii) Exceptional program executions.

This thesis is primarily concerned with the extension of the LARVA tool and logic with statistical capabilities. This is achieved through the creation of a statistical framework operating on top of the current logic, with the result being a runtime verification tool capable of expressing statistics, henceforth called LarvaStat. The creation of LarvaStat involves a number of steps, the first being the conceptualisation of the core notions driving the statistical framework. Following this, we also present a formal analysis of the resulting framework through the specification of a formal semantics expressing the statistical logic. With the theory complete, we present a domain specific embedded language extending LARVA, thus offering a medium for the expression of statistics. The final step involves the implementation of a compiler, which converts statistics expressed using the defined language into an executable artefact. Finally, we also present a comprehensive case study for the justification of LarvaStat based on Intrusion Detection.

Contents

1. Introduction

1.1 Overview

Ever since the inception of software development, there has been the requirement of a method which offers guarantees of the system's correctness. Moreover, as computing becomes increasingly ubiquitous, and as computer systems influence our lives in more frequent and crucial ways, the need for reliable software grows accordingly. It is a practical fact that faults, especially faults within critical systems — be it a money transaction processing system or the software found within medical equipment — can be costly, potentially costing time, money and even human life.

Any software artefact is required to behave in the manner originally intended by its developers, and it is these faults which deviate system behaviour from that required of the system. Hence, the achievement of software correctness can be considered as the act of ensuring that the system implementation adheres to its corresponding specification, with the ultimate holy grail being a guarantee of the absence of bugs. Currently, the most common approach for this achievement is through testing. Although testing is scalable and can be effective if intelligently applied, it inherently lacks coverage. In other words, as Dijkstra once famously said "testing shows the presence, not the absence of bugs", whereby a system can be tested under a million different scenarios, and is still susceptible to breaking on the million-and-first test case.

Another approach for the guarantees of correct software is through the use of formal verification techniques, most notably model checking. In this case, we try to verify the correctness for each possible execution trace available to the system, usually through the use of mathematical techniques. Whereas model checking solves the issue of coverage and offers the best guarantees, it is not scalable to any system of appreciable size making it of little practical use. Moreover, model checking is usually a non-trivial task, requiring a considerably strenuous effort to apply. In truth, another criticism leveled at both approaches the fact that neither take the runtime environment into serious consideration. It is all well and good to verify the system under controlled or theoretical environments,

however in practice the system environment is dynamic, unpredictable and varied, making the verification of software under typical scenarios faced beyond deployment difficult.

We therefore require an alternate manner with which to ensure software correctness. This approach, commonly known as runtime verification, verifies the currently executing system trace (during the actual system execution) for adherence to the specified required behaviour. Hence, while guaranteeing coverage, this approach also monitors the system behaviour as it is affected by the underlying environment. Runtime verification can be viewed as an extension to testing with augmented guarantees through the use of more powerful specification languages, and admits a set of crucial issues.

One such core issue is the choice of language for the expression of system requirements. In other words, we require a means with which to express the required correct system behaviour. Such languages are either automaton-based or logic-based, whereby all approaches focus on the verification of natural occurrences within the system. Such occurrences include event sequentiality, property invariance as well as temporal aspects. In particular, real time (temporal) properties — event A must not occur more than twice per minute — typically form a considerable subset of the required specifications which need to be verified at runtime. Thus, choosing a language with real time expressivity crucial. It is for this reason that historically temporal logics and automata augmented with the notion of time (such as timed automata [2]) have been frequently used for runtime verification.

Directly bound to the choice of language are the issues of monitoring, instrumentation, system overhead and reparation. Given a set of specifications which the system is to abide to, we require a mechanism for their verification. This is achieved through an executable monitor running in parallel with the underlying system under scrutiny. Hence, we require an executable means with which to operate system requirements within the monitor. In fact, whereas automaton-based languages are directly executable, logic-based approaches usually require the development of a monitoring algorithm for their execution, or possibly the conversion to a semantically equivalent executable logic. However, apart from the execution of specifications, we require the instrumentation of the monitor with the underlying system, thus giving monitors access to required system information. Such information is crucial for the monitor to deduce whether the current system system behaviour is classified as correct or not.

Given that this monitor responsible for runtime verification is executed in parallel with the underlying system, this implies that certain resources (both computational as well as memory resources) are consumed by the monitor otherwise available to the system. It is hence crucial to reduce this overhead to a minimum, especially since the introduction of runtime monitors can alter the system behaviour itself, possibly breaking properties

which are otherwise adhered to. This overhead can become considerable unless properly handled, whereby in general there exists a tradeoff between logic expressivity and system overhead. Hence, choosing a very expressive language, although desirable for the intuitive expression of specifications, may incur a considerable overhead during runtime.

Whereas we have established that specifications should be written for the characterisation of correct (or incorrect) system behaviour, it remains unclear what should be done once incorrect behaviour is identified. In fact, whereas certain languages permit only the characterisation of incorrect behaviour, others (such as DATEs [12]) go one step further, by allowing for the execution of reparatory actions in order to steer the system back into an acceptable state. If for example we specify a property which disallows three attempted bad logins from the same ip, an acceptable reparatory action upon the breaking of this property is the blacklisting of the culpable ip for a specified amount of time.

Although runtime verification is an exciting relatively new approach to software verification which offers interesting results, it is non-trivial as highlighted by the core issues identified above. In practice, the choice for the attempted solution to such issues boils down to the application domain and requirements. If for example the application places emphases on real time response times or is executing within an environment with limited resources, runtime verification may not be the most adept approach, or perhaps at most only use simple languages which imply little overhead. However, if this application is of mission critical nature or with high security requirements, where incorrect results or uncaught intrusions are very costly, the application of runtime verification techniques are ideal.

We theorise that the augmentation of runtime verification with statistical capabilities is advantageous. Firstly, it is often better to watch for indicators of impending failure than to wait (for failure). The specification "The program should not contain memory leaks" is of little use, since by the time the specification is broken the program would have crashed. This specification can be reworded into "The count of memory allocations and de-allocations should be equal", whereby through such a specification the impending memory leak can be caught prior to the program crashing, taking the necessary reparatory action accordingly. Moreover, statistics allow for the straightforward expression of non-functional system requirements, such as specifications relating to performance and reliability. Whereas most current logics used for runtime verification focus on specifying what the system should do ("event A should occur after B"), online statistics collection allows for the quantification of how the system is, and can hence be applied to fields such as performance profiling and quality of service, which leads us to the next point. From a utilitarian point of view, the collection of statistics over runtime executions admits vast applications. Fields benefitting from such functionality include program profiling, user modeling, intrusion detection, performance profiling and quality of service. Finally, it

is worth pointing out that the augmentation of certain logics with statistical capabilities may also enhance the logic expressivity. If nothing else, the addition of statistics certainly increases the conciseness and intuitive expression of certain specifications. Properties such as "the count of event A should not exceed 3" inherently implies the need for statistical computation (the counting of an event), and hence such a property is more easily expressible through statistics.

The augmentation of runtime verification with the notion of statistics implies the need for an underlying runtime verification framework. We identify this framework to be LARVA [12] (standing for Logical Automata for Runtime Verification and Analysis), an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs believed to be sufficiently expressive for our requirements. Based on Dynamic Automata with Timers and Events (DATEs) as the underlying mathematical framework, LARVA allows for the expression of (i) Temporal aspects dealing with consequentiality (event A must occur before B) and real time (Event A must occur nor more than twice every five minutes) (ii) contextual aspects with the possibility of monitoring objects either globally, grouped according to their container or individually (Every account must belong to a registered user) (iii) Exceptional program executions.

This thesis is primarily concerned with the extension of the LARVA tool and logic used for runtime verification with statistical capabilities. We introduce such statistical capabilities to LARVA through the creation of a statistical framework operating on top of the current logic, with the result being a runtime verification tool capable of expressing statistics, henceforth called LarvaStat. The creation of this framework and tool is dissected into a number of steps. The first step involves the conceptualisation of the core notions driving this framework, and involve a mixture of theory and practice. Whereas the designed constructs are dictated by practical requirements in the above mentioned fields, their expressivity and execution is bound by the mathematical framework driving LARVA. Following this, we also present a formal analysis of the resulting framework through the specification of a formal semantics expressing the required statistical logic. With the theory driving the statistical framework complete, we present a domain specific embedded language extending LARVA, thus allowing for the practical expression of the required statistics. Hence, the final step includes the implementation of a compiler, which given the expression of a set of statistics (using the defined language) produces an executable artefact which implements the required specified statistical logic. Note that as a justification to the creation of LarvaStat, we also present a comprehensive list of case studies, as well another major case study (introduced in the next section).

1.2 Broad Aims

As previously stated, we require the conceptualisation and development of a statistical framework operating on top of LARVA which enables the collection of statistics over runtime executions. We identify a set of broad aims which we aim to achieve through this statistical framework which are enlisted below.

- To create a statistical framework capable of collecting a wide variety of statistics over runtime executions, in such a manner which is intuitive and straightforward to the user (of this framework), while keeping the resulting system overhead to the possible minimum.
- To study the relationship between runtime verification and statistical frameworks, with the aim of devising a fruitful integration of both into one complete framework whose overall expressivity, applicability and overall suitability is greater than the sum of its constituent parts.

Following the creation of this statistical framework we aim to prove its achievement of the above aims. Hence, we present a set of interesting case studies, however more importantly we shall implement a major case study primed as the major justification to the creation of LarvaStat, and is essentially the implementation of a probabilistic intrusion detection system and integrated system profiler, while also applying statistical anomaly detection techniques. It is for this reason that the field of intrusion detection, in particular statistical anomaly detection, shall later also be introduced in some detail.

1.3 Document Outline

The following dissertation is organised into four main parts as follows:

- The first part introduces the necessary background for the comprehension of the remainder of the presented work. Chapter 2 discusses in some detail the notion of runtime verification. Chapter 3 presents the tool and logic LARVA, as well as the underlying mathematical framework based on DATEs. Finally, chapter 4 thoroughly motivates the augmentation of runtime verification with statistics, an also presents a comprehensive overview of the current approaches to the field.
- The second part describes LarvaStat in all its entirety. Chapter 5 introduces and motivates the notions driving our statistical framework. Chapter 6 explains the technique used for the execution of the statistical framework. Chapter 7 proposes the design of a domain specific embedded language extending LARVA with statistical capabilities, and finally chapter 8 describes the compiler implementation which given a statistical framework expressed using the designed language, produces an executable artefact.

- The third part analyses and evaluates the statistical framework. Chapter 9 formally specifies the core notions defined for the framework, placing special emphasis on the notion of the interval as well as specifying an operational semantics for the framework. Chapter 10 introduces the practical side of LarvaStat through the discussion of a few select real life scenarios. Chapter 11 introduces the field of intrusion detection, placing emphasis on statistical techniques used in the field. Moreover, this chapter discusses the difficulty with intrusion detection and concludes by discussing the major case study implementing a probabilistic intrusion detection system and integrated system profiler.
- The final part concludes the work presented in this thesis.

Part I

Background & Motivation

2. Runtime Verification

Given the lack of coverage implied by testing and the intractability of exhaustively verifying software, an alternate approach whose concern is the verification of the currently executing trace is gathering attention. This technique is known as runtime verification, and can be viewed as an extension to testing with augmented guarantees through the use of more powerful specification languages. The following chapter aims at introducing this field, with section 2.1 motivating and introducing the core concepts defining runtime verification, and sections 2.2, 2.3, 2.4, 2.5 and 2.6 discussing these core concepts in more detail. The chapter concludes by giving an overview of currently available tools in runtime verification.

2.1 Overview

Ever since the inception of software development, there has been the requirement of a method which offers guarantees of the system's correctness. In other words, this methods is to ensure that the implementation conforms to its corresponding specification, with the ultimate holy grail being the guarantee of an absence of bugs. The field which studies the development and application of such a method is called software verification, and admits multiple techniques verifying system correctness on various representations of the system. Whereas some [12, 17, 14, 13, 11] classify these techniques on the basis of whether they are applied before or after the system execution, we present a more insightful classification discussed below. Concisely, we believe software verification techniques to fall under one of three categories, these being static analysis, dynamic analysis and the verification of select traces.

By static analysis we refer to all verification techniques applied directly to the system code. Such techniques are usually used to identify certain superficial inconsistencies with the code, such as potential infinite loops or memory leaks, however in truth are incapable of identifying more complex semantic errors. Techniques which fall under the static analysis category include control flow graphs [24] (allowing for superficial code analysis such as a reachability analysis) and the calculation of code metrics.

Dynamic analysis techniques refer to a class of verification techniques which operate on a representation of the semantics implied by the system code prior to execution. Hence, dynamic analysis techniques mostly refer to formal verification techniques such as theorem proving [33] and model checking [10]. Such techniques (model checking in particular) focus on formally proving software correctness by proving adherence of each possible program trace to the required formally defined program specifications. However, although such techniques offer the greatest guarantees on all event traces prior to execution, a currently insurmountable issue is the fact that they do not scale well to any system of appreciable size. Moreover, another issue with such techniques is the fact that proving the semantics to be correct (with respect to the specification) does not imply the implementation to be correct. In other words, even if we are somehow given assurances that the semantics driving the system are correct, the implementation of such semantics may introduce new problems. This problem is made worse by the introduction of the environment which the system is to execute in, since the environment may alter the system behaviour too.

The third classification includes any verification technique applied on the system execution. Hence, given that each program execution can traverse only one of its (numerous) possible event traces, such techniques are considered to be techniques verifying select event traces, and can be executed online (in parallel with the system execution) or a posteriori (after the execution). Techniques which fall under this category primarily include testing and runtime verification. Testing is the most commonly used verification technique in industry, and involves checking a (very small) subset of all possible event traces for correctness, whereby these chosen event traces are believed to represent situations of interest. The main issue with testing is the systematic intelligent generation of these event traces. In fact, testing techniques such as acceptance testing, regression testing and stress testing aim at executing the system under different test case scenarios of interest. Nevertheless, testing can always only find the presence of bugs and not the absence, which implies that testing inherently suffers from a lack of coverage. Moreover, testing finishes upon delivery of the system, which implies that the system is unguarded against errors once it has gone live.

In contrast, runtime verification verifies that the event trace generated at runtime by the system adheres to the specification. Hence, the issue of event trace generation is absent for runtime verification techniques, since the event trace is generated by the system execution itself. Verification is carried out by an executable oracle (monitor), and although runtime verification is traditionally executed in parallel with the system execution (online monitoring), runtime verification can also be executed a posteriori (offline monitoring) mostly for reasons of monitor overhead — more below. Monitoring of the current system execution can go beyond system deployment, which implies that runtime

verification can offer protection against errors even after deployment. Concisely, runtime verification can be viewed as an extension to testing, since it offers higher guarantees without the issue of a lack of coverage implied by testing. As specified in [14], this increased level of guarantees is obtained through the use of more powerful specification languages. In truth, runtime verification is the preferred choice for systems with strong requirements on criticality, reliability and security [11]. Below is a general scenario defining runtime verification.

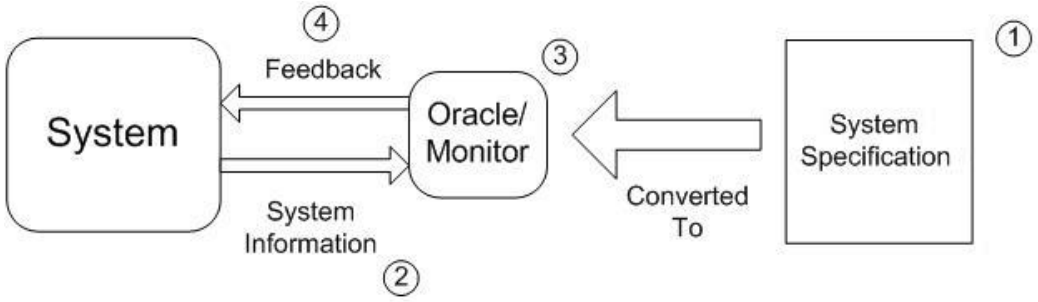


Figure 2.1: Runtime Verification Overview

1. **System specification** — The very first step in the deployment of a runtime verification framework, whereby correct system behaviour which the oracle is to monitor the underlying system for adherence to is specified. In practice, complete specifications are rarely available, and hence it is unrealistic to assume that every possible system behaviour can be characterised as correct or failed. Nevertheless, it is still possible to specify critical aspects of interest within the underlying system [11]. Also, implicit in this step is the requirement of a language used for the expression of these specifications. In general, no language is best and the choice of language often depends on multiple factors such as the domain the system is applied to, as well as the available computational resources.

2. **System Information** — For the monitor to be able to verify correct behaviour, it requires a stream of information describing the system behaviour such that verification is executed on this stream. System information is primarily represented in either of two forms, these being state based (as specified in [6]) or event based (as specified in [13]). An issue tightly bound with the representation of system information is instrumentation, which refers to the executable mechanism which actually retrieves this information and passes it to the monitor. Instrumentation can either be automated (using techniques such as aspect oriented programming [26]) or manually developed by the user.

3. **Oracle/Monitor** — Given a system specification, this specification is to be converted to an executable monitor verifying that the system specification is adhered

to. This monitor can either be executed in parallel with the system execution (online verification) or a representation of the executed system trace stored, and the monitor executed after the system has executed (offline verification). In the case of online verification, the monitor itself must be streamlined in order to pose as little overhead on the system as possible.

4. **Feedback** — In certain cases, a property specification is associated with a reparatory action. Hence, in the case that the system violates this property, this action can be executed in order to steer the system back to an acceptable state. Obviously, the option of executing a reparatory action is only available for online monitoring, since the system would have long executed past the property violation when monitoring is executed offline.

The remainder of the chapter focuses on the crucial issues constituting runtime verification introduced above. On a final note it is worth mentioning that, as discussed in [12], the divide in techniques for software verification is often overemphasized, whereby in fact it is advantageous to use multiple techniques in conjunction. An example scenario where techniques from different classifications is used is the following, whereby the first few states are model checked for correctness (and is possible due to the simplicity and relatively small size when considering only the first few states), and the remainder of the traces are verified at runtime. In other words, model check the system until the problem becomes too large, then switch to runtime verification, which is scalable. Another example is the use of runtime verification in conjunction with static analysis techniques, since static code analysis can in certain ways act as a predictive technique declaring only a subset of possible choices given that the user chooses a particular execution path at runtime.

2.2 Languages for Requirements Specification

The notion of runtime verification implies the need for a means with which to express properties which the system is to adhere to. These properties specified in the appropriate language are consequently to be converted to an executable monitor, which implies that this language is to have an associated operational semantics capable of implementation and execution. With execution comes system overhead, since the monitor requires additional memory and computation time which could have been used by the system itself. Hence, any monitor is to be as streamlined in order to as little overhead as possible. In fact, from the point of view of the choice of language (for runtime verification), the issue of the language execution is crucial, since there exists a tradeoff between language expressivity and system overhead. In other words, expressivity is computationally expensive, and making a language very expressive results in additional overhead imposed on the system.

In general no language can be considered the best, and in general some languages are better to express certain properties than others. Often the choice of language often depends on a number of factors, some of the most important being the application domain, language expressivity and the implied overhead. If for example the expression of a property requires the notion of time (event A should not occur more than three times every minute), we require a temporal logic such as linear temporal logic [10]. If on the other hand we require the expression of sequentiality (event A must occur after B), extended regular expressions may suffice. In general, languages used for runtime verification can be classified under two major approaches, these being automata based and logic based approaches, as shall be discussed below.

2.2.1 Automata in Runtime Verification

The use of automata based languages for the expression of properties within the runtime verification framework has a long history [12, 2, 18]. The most crucial advantage with using automata to describe system properties is the inherently intuitive nature implicit with the structure. Moreover, automata can be easily visualised and manipulated, which is a good virtue to have especially given the previously motivated requirement of intuitive and straightforward property definition. Below is an example timed automata, which augments the notion of an automaton with time.

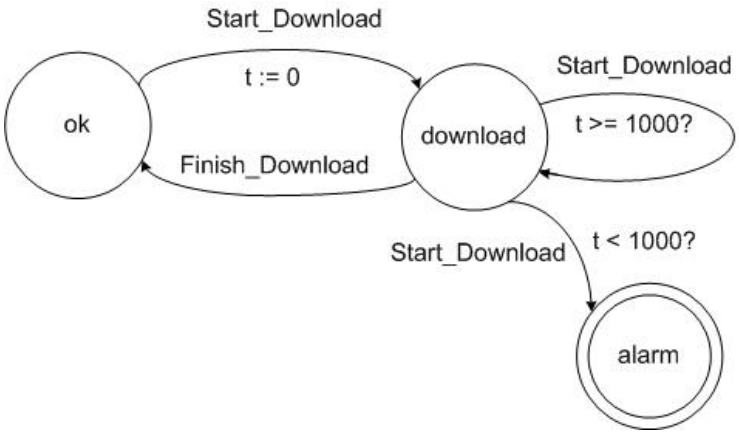


Figure 2.2: An example timed automaton.

As can be seen above, it is immediately apparent that the above property is specifying that no other download may start prior to the elapsing of one second since the last download. Specifying such a property using other formalisms would most probably not be as intuitive. In practice, automata allow for the specification of a wide variety of properties, and handles well issues such as sequentiality and condition. Moreover, automata handle the augmentation of the core concept with additional features well, as evidenced by augmentation of time in timed automata (above) and the augmentation of communicating

automata in DATEs [12]. On the other hand, issues such as interval and duration may not be intuitively specified using automata, and would be better handled by other logics such as duration calculus [8] and regular expressions. In truth, the applicability of automata is dictated by the specifications we wish to specify.

2.2.2 Logics in Runtime Verification

Logic has been studied for a long time, long before software verification was created. This lead to the development of numerous logical frameworks, such as deontic and temporal logic, which formally reason about distinct natural occurrences. Whereas deontic logic reasons about issues such as obligation and necessity (and is therefore suitable for practical applications such as contract analysis), temporal logic represents and formally reasons about the notion of time. Hence, given the inherently temporal nature of software execution it is for this reason that temporal logic has been widely used for verification purposes. In fact, numerous temporal logic based approaches have been presented throughout the literature for the expression of system properties, each presenting alterations and improvements to previous notations. An example temporal logic which has been particularly studied, extended and applied to runtime verification scenarios is linear temporal logic [10], and allows for the expression of properties such as "condition A will eventually be true" or "condition A is always true". It may also specify properties such as "condition A is true until condition B becomes true". Note that (as will be discussed below), the choice of logics for the expression of system properties also depends on the computational overhead induced by executing the logic. Also, an interesting issue worth mentioning is that certain logics go beyond the expression of properties, by essentially providing a meta-language capable of expressing other logics. An example of such a logic is that defined for EAGLE [6], whereby by using a concise base logic, other logics (such as linear temporal logic) can be encoded within the base logic.

We identify three issues with respect to applying temporal logic to runtime verification scenarios, these being complete vs incomplete trace, infinite vs finite trace and future vs past logics. Logics which assume a complete trace are capable of accessing all of the trace at any instant. Hence, logics requiring a complete trace are only applicable to offline monitoring. On the other hand, logics which assume an incomplete trace mirror the fact that the trace is being generated on the fly during online monitoring. Hence, logics assuming an incomplete trace add certain complexity for the evaluation of properties requiring knowledge of the future or liveness properties, since certain properties remain unresolved until additional trace information is generated. Another issue is that of infinite vs finite trace logics, whereby certain logics assume the trace to be infinite (such as Buuchi automata), whereas others apply logic to a finite trace (such as interval temporal logic [5]). In practice, the use of infinite traces mirrors the situation of a continually executing system. The remaining issue is whether the logic allows for the expression of properties referring to time instances in the future or the past. Such example properties could be

"event A should occur in the next five time units" or "given the occurrence of event A, event B should have occurred three time units ago", and although such expressivity admits copious practical applications, it does not come without a computational expense. Whereas properties requiring future valuations remain unresolved until these valuations become available, past logics require memory of the past event trace up till a certain past instant in time.

2.2.3 Time Models

Any defined temporal logic or automata-like structure augmented with the temporal functionality implies a model representing the notion of time driving the logic (we shall be focusing on logics used for runtime verification). Multiple models exist, whereby we identify three distinct classifications of such time models. These classifications are dense vs discrete, implicit vs explicit and point vs interval time models, and are discussed below. Moreover, classifications can be combined in order to present specific time models. For example, duration calculus [8] is a temporal logic whose implied time model is implicit, interval based and dense. Each possible combination implies a distinct approach towards the simulation of time, and hence each approach implies different expressive power and are of different computation complexities.

Dense vs Discrete Time Model

Dense time models imply that time is represented by (the positive subset of) the real number line. Although the use of dense time models for the representation of time results in certain paradoxes such as zeno-like behaviour, a continuous time line also has valid practical applications. In truth, the use of dense time models presents issues regarding the computability and complexity of the time model, as well as the practical computation of such representations of time (due to the inherently discrete nature of a computer). Hence, solutions to such problems are presented, most notably that of digitization whereby the real time number line is converted to the discrete number line [12]. An example temporal logic which employs a dense time model is duration calculus [8]. On the other hand, dense time models represent time using discrete steps. Discrete time models can be efficiently computed, however whereas certain applications are satisfied by employing a discrete time model, other applications require a dense time model. An example temporal logic using a discrete time model is linear temporal logic [10].

Implicit vs Explicit Time Model

An explicit time model explicitly encodes time within the logic. Hence, for example the property "the response time should not exceed three time units" is encoded as $\forall\, t_1, t_2 \bullet \text{input}(t_1) \Rightarrow \text{output}(t_2) \wedge ((t_2 - t_1) \leq 3)$. Note how time is explicitly encoded in t_1 and t_2. An example formalism using an explicit time model is DATEs [12], whereby the triggering

of a time event after four seconds is declared as timer@4, thus explicitly defining the time. On the other hand, an implicit time model never directly refers to the time within the logic, however the logic is implied within the logic constructs. An example logic employing an implicit time model is linear temporal logic [10].

Point vs Interval Time Model

A logic can either express a property which is evaluated on a specific point in time, or over a sequence of points (an interval). Clearly no approach is best, and the use of a point or interval depends on the property we wish to express. Hence, a property such as "Check that the user has the necessary rights each time a file is manipulated" requires the checking of the property at each point where a file is manipulated, whereas the property "check that the number of downloads within the last hour does not exceed five" implies the requirement of an evaluation on all the points within the last hour. Note that the introduction of intervals may introduce additional computational complexity, especially with regards to space complexity since some form of partial interval representation is possibly required. Also, as specified in [12], specifying intervals on a dense time model is undecidable due to the number of sub-intervals within that interval being equal to the number of all possible subsets within that interval. An example logic based on a point time model is linear temporal logic (although LTL expressions may refer to values in the past or future, an LTL expression always returns the valuation on a particular point in time), whereas a logic based on an interval time model is duration calculus.

2.3 Online vs Offline Monitoring

The currently executing system execution is verified by an instrumented monitor. Moreover, this act of verification can either be executed in parallel with the system execution or on a stored representation of the executed system trace post the actual system execution. In fact, the choice of whether to monitor online or offline impacts comprehensively the allowed logic expressivity (and resulting overhead), affects the issue of reparation and also comprehensively impacts the nature of verification as discussed below.

Online monitoring implies that the monitor consumes computational resources otherwise available to the system. This implies that an online monitor should be as streamlined as possible, since we want to reduce the overhead implied (which can be substantial) to an acceptable level — more below. However, this issue is absent for offline monitoring, since offline monitoring is executed post the actual system execution, without consuming any resources otherwise available to the system. On the other hand, offline monitoring implies that any property violation which might have occurred would have been long gone by the time the violation is detected, hence restricting the monitor from possibly executing any corrective action. This issue is absent from online monitoring, since property violations

are caught in a timely fashion, allowing for any necessary reparatory actions to be executed. Another issue worth mentioning is that online and offline monitoring differ at the conceptual level. Whereas offline monitoring is usually used for testing purposes [11], and is mostly applied prior to deployment or for log analysis, online monitoring can be used for the implementation of corrective behaviour, even as part of the system development itself. In fact, some [12] suggest that runtime verification should be ingrained along the development life cycle. This notion is also extended to programming paradigms, giving rise to monitoring-oriented programming. Finally, recall the distinction previously made for logics whose trace is complete, and logics whose trace is incomplete and generated on the fly. This distinction is directly mapped to logics applied for offline and online monitoring respectively.

2.4 Instrumentation

For the monitor to be able to verify the system behaviour, we require a mechanism which allows access of system information to the monitor. This communication between system and monitor is achieved through the installation of the monitor with the target system, running within the same environment however distinct from the system itself [11]. This installation is commonly known as monitor instrumentation, and can be done either manually or in an automated fashion. However, given that manual instrumentation is prone to errors, we shall focus on currently available techniques for automatic instrumentation.

Multiple techniques and technologies exist for the achievement of automatic instrumentation, whose applicability often depends on the language used for the system implementation, and possibly also on the environment the system is executing within. A commonly used technique for the achievement of instrumentation is through the definition of points within the source code where the monitor is instrumented i.e. execution control is passed to the monitor at these specified points so that the monitor can execute the necessary algorithms checking for correct system behaviour. While valid, this approach requires a substantial amount of work since specifying each required instrumentation point within the code involves a lot of work. Another commonly used technique is the alteration and manipulation of the environment where both the system and the monitor are executing. This technique is exemplified in [35], whereby the underlying operating system kernel is modified so that monitors have control over the currently executing system calls within the system. However, we shall be focusing on the application of aspect oriented programming techniques [26] for instrumentation, since it is through this technology that LARVA [12] (the runtime verification tool of our choice for the augmentation with statistical functionality) achieves automatic instrumentation.

2.4.1 Aspect Oriented Programming

At the core of aspect oriented programming [26] lies the notion of crosscutting concerns. Imagine the implementation of a transaction system which defines multiple components, these being the core transaction processing module, a security module, an information storage module and a logging module. It stands to reason that activities occurring in the security, information storage and transaction processing modules need to be logged. However, this implies that whereas the first three system components (called concerns within aspect oriented terminology) can be implemented in a modular fashion, the logging module is said to crosscut all three modules, since the logging component requires its implementation to be spread over the other modules. Hence, the installation of a new augmented logging component would require the alteration of all other three components due to the crosscutting nature of the logging component. Aspect oriented programming allows for the implementation of all four components in modular fashion, thus increasing the system implementation modularity by allowing for the separation of concerns.

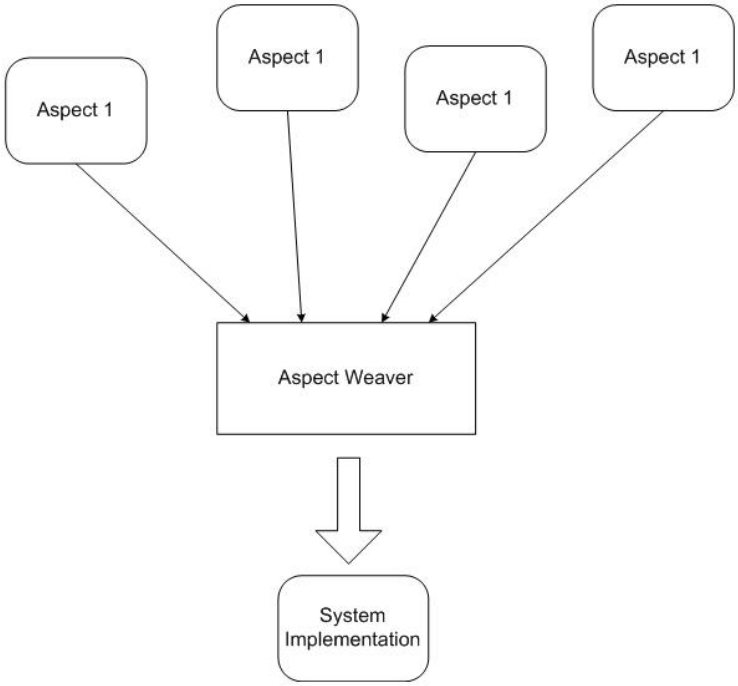


Figure 2.3: An overview of AOP. Based on [26]

Each concern is implemented in a modular fashion using traditional object oriented techniques. Consequently, all concerns are fed to an aspect weaver which integrates all concerns into one final system implementation.

Perhaps one of the most popular implementation of aspect oriented programming is AspectJ [25], a Java implementation whose core construct is the aspect, implemented

through pointcuts and the advices. Pointcuts allow for the definition of join points, points within a program execution where multiple aspects are integrated. Pointcuts can define a variety of such execution points, such as method calls, constructor calls as well as the throwing or handling of exceptions. An example pointcut is the following

```
pointcut startDownload() : execution(* startDownload(..) );
```

whereby pointcut startDownload() specifies all execution points within the the underlying system where method call startDownload is called.

With the declaration of pointcuts, the next step is to declare advice upon the triggering of pointcuts. This advice takes the form of Java code and can be declared to execute *before*, *after* or *around* a join point declared by a pointcut, as exemplified below

```
before() : startDownload() {
    System.out.println("Download Started!");
}
```

The above advice specifies that the comment "Download Started!" is output to console before any download is started. For more information regarding AspectJ syntax and semantics please refer to [25].

The functionality broadly described above offers a perfect platform for automated instrumentation. Using pointcuts (or their equivalent in other aspect oriented programming implementations) we can define particular points within the underlying system code which specifies where the underlying system code is to be integrated with the runtime verification monitor. Hence, all that is required is the implementation of a monitor which simulates the specified properties expressed using a particular logic, and an aspect which instruments this monitor with the system. In truth, the use of aspect oriented programming for automated instrumentation has been used considerably, and can be seen in [13, 12]. In the case of LARVA (discussed in detail in the following chapter), the LARVA compiler outputs both the Java code for the simulation of the monitor, as well sa the aspect which instruments this monitor with the underlying system.

2.5 Reparation

In the general case of a system whose implementation is imperfect (admits a set of faults), the said system will eventually stumble in an unwanted state i.e. a state which is not allowed by the system specification. It is the job of the runtime verification monitor to notice such an event. However, whereas certain logics and tools are content with simply capturing and notifying the user of such an event (such as LOLA [14] — more below), it is often possible to take the necessary reparatory action so as to steer the

system back into an acceptable state. Imagine the property "a user should not be allowed more than three consecutive bad logins". An acceptable action acting as reparation would be the blacklisting of the user for an hour post the breaking of the property. An example logic allowing for reparation is DATEs [12], an automaton based logic (more below) whereby each transition is associated with an action, and each time the transition is traversed the action is executed. The need for reparation usually depends on the property being specified, although experience shows that the feature of allowing reparation is often required.

2.6 System Overhead

Runtime verification implies the requirement of a monitor which checks that the system is adhering to the specified properties describing correct behaviour. Whereas in the case of offline monitoring the issue of system overhead is avoided (since verification is executed a posteriori to the actual execution, possibly at a different site), online monitoring invariably implies that the monitor is to execute in parallel with the system, thus consuming resources otherwise available to the system. In fact, experiments conducted on case studies show that this imposed overhead can be quite considerable [12].

In general, the overhead implied by online monitoring cannot be completely removed. However, it is crucial that this overhead is reduced as much as possible till it reaches an understandable level. In practice, this level is dictated by the system requirements and its executing environment. Hence, a system with substantial resources whose nature is not in any way critical may afford more overhead than one of critical nature with real time response requirements and executing with limited resources. Nevertheless, we should still aim at reducing the overhead as much as possible, mostly by using efficient instrumentation techniques as well as by avoiding complex computations.

A valid issue worth mentioning is the fact that given that monitoring uses resources (in the form of memory and computation), this resource consumption may actually alter the system behaviour. In other words, we can essentially apply the Heisenberg uncertainty principle to online monitoring; by observing the system we are actually altering its behaviour. This implies that a system may behave differently given more time or memory, such as executing faster or more efficiently. Hence, online monitoring does not actually monitor the underlying system, but in fact monitors the system affected by the instrumented monitor itself. Certain situations may arise where a property is broken which would not have been had the monitor not been present. Conversely, there is also the possibility of a property which holds given the presence of the monitor, and otherwise breaks once the monitor is removed. Note that this issue is being studied by characterising sets of properties which are said to be speedup or slowdown invariant i.e. the property still holds even if the system is slowed down / sped up. Such characterisation has already

been presented for properties written using duration calculus [12], and another paper is currently submitted for publication regarding such issues for timed regular expressions.

2.7 Runtime Verification Tools

A wide variety of tools are available for the implementation of runtime verification monitors. The following entails a list of such tools

- Java-MaC [27] defines two specification languages with the aim of integrating implementation with monitoring and specification. These two specifications are called PEDL and MEDL, whereby PEDL is used to define primitive events which occur within the system, and MEDL is a high level specification language for the specification of system properties.

- EAGLE [6] is a language independent tool and rule based logic used for runtime verification. Although the logic defined is parsimonious, it in fact is sufficiently expressive to encode multiple formalisms within the same logic, such as linear temporal logic, interval logics, finite state automata, extended regular expressions and is also sufficiently expressive to encode statistical properties. Hence, the EAGLE logic is often also considered as a meta-logic, capable of defining other logics. An advantage of this approach is that all encoded logics can be used in conjunction within the same script. Note that Hawk is a programming oriented extension to EAGLE implemented for the Java language.

- LOLA [14] is a is a functional stream computation language, similar in nature to synchronous languages such as LUSTRE [19] and allows for the specification of properties both in the past and in the future. While also allowing for the specification of properties both in the past and the future, LOLA represents one of the first approaches which explicitly attempts to augment runtime verification with statistical capabilities. Another interesting feature of LOLA is that given its similarity to synchronous languages it offers guarantees regarding memory requirements prior to the actual execution.

- LARVA [12] is an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs. Unlike previous approaches (which were logic based), LARVA offers an automaton based approach towards the specification and monitoring of system properties. Moreover, apart from being considerably expressive with real time functionality and allowing the communication of automata, another interesting feature is the specification of contextual properties. By contextual properties we imply properties which we would like to monitor on a per entity basis, such as “Each user should not be allowed access unless the proper credentials are assigned”. Hence, in this case the process of monitoring access should be replicated for each user logged in.

- Java-MOP [9] is a Java implementation for the monitoring oriented programming paradigm, whereby the act of system and monitor implementation are integrated into one development phase. This implies that monitoring can go beyond runtime verification purposes, by implementing the monitor as part of the system design rather than only for the verification of correct behaviour. Moreover, as stated in [12], Java-MOP can be extended with different logics including future time and past time linear temporal logic, as well as extended regular expressions.

The augmentation of runtime verification with the notion of statistics, which is the aim of this thesis, implies the need for an underlying runtime verification framework. We identify this framework to be LARVA, since not only is the logic believed to be sufficiently expressive for our requirements (more below), but also due to external factors allowing for complete access to the LARVA source code, as well as having direct contact with the developers which was crucial for the development of our statistical framework. Note that the above list is in no way complete, whereby numerous other tools such as Temporal Rover [20] and ConSpec [1] have been left out. For a more complete overview of available tools refer to [12]. Hence, note that henceforth we shall be focusing on three tools, these being EAGLE, LOLA and LARVA. The first two shall be analysed in detail in chapter 4 due to being the only tools encountered which allow for the specification of statistical measures (directly or indirectly), whereas LARVA shall be thoroughly discussed in the following chapter due to being the runtime verification framework which we shall augment with statistical capabilities.

2.8 Conclusions

Runtime verification offers an alternative approach to software verification which avoids the intractability of formal verification, without compromising coverage implied by testing. Nevertheless, the field is non-trivial and entails a set of problems, including the choice of language for expression of the required system properties, as well as the minimisation of monitor overhead. In truth, runtime verification can be applied to various scenarios and allows for extensive modification capabilities. If for example we are interested in using the field for testing purposes, the system trace can be verified through offline monitoring, hence implying that the issue of overhead is avoided. However, offline monitoring also implies the lack of reparation. On the other hand, if we wish to apply runtime verification to critical systems handling vital resources (such as time, money and even our life), this implies that online monitoring is more suitable, so that if the system behaves in an unexpected fashion the monitor can take the necessary reparatory action. However online monitoring implies overhead. In truth, we believe work on runtime verification to be far from complete, and requires substantial further research in order to achieve a higher level of maturity. Such research is presented in this thesis, through the investigation of the impact of augmenting intrusion detection with statistical capabilities.

3. LARVA

The augmentation of runtime verification with the notion of statistics implies the need for an underlying runtime verification framework. We identify this framework to be LARVA [12] (standing for Logical Automata for Runtime Verification and Analysis), an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs believed to be sufficiently expressive for our requirements. Based on Dynamic Automata with Timers and Events (DATEs) as the underlying mathematical framework, LARVA allows for the expression of (i) Temporal aspects dealing with consequentiality (event A must occur before B) and real time (Event A must occur nor more than twice every five minutes) (ii) contextual aspects with the possibility of monitoring objects either globally, grouped according to their container or individually (Every account must belong to a registered user) (iii) Exceptional program executions. The following chapter involves a broad overview of the LARVA framework (section 3.1), the definition of the DATE mathematical framework (section 3.2) which LARVA is based upon and a detailed discussion of the LARVA script (section 3.3) which serves as the medium through which system properties are expressed. Note that the following chapter contents are based on the work presented in [12], and hence we make no claim of the following ideas being our own invention (apart from where noted).

3.1 Overview

LARVA is essentially a tool and automaton-based logic used for runtime verification which enables users to verify adherence to crucial system properties. Given that LARVA is based on DATEs as the underlying mathematical framework, these properties are to be expressed as DATEs; automaton-based constructs with additional real-time functionality (through the use of timers) and automaton communication (through the availability of channels). Moreover, DATEs are associated with a context (in the form of an object), such that on each new context creation triggers a corresponding new DATE instance. This allows for the specification of contextual properties, either on a global level or on a per object basis.

A user is to provide two components, these being

- The system implementation
- The LARVA script

Both implemented components are interdependent. Whereas the system implementation requires adherence to a set of specified properties, it is the LARVA script that defines and verifies these properties through the instrumentation of an executable monitor. Hence, we require the LARVA script to define (i) the system properties expressed using DATEs, and (ii) information required for the instrumentation with the underlying system. A LARVA script specifies a set of DATEs, timers and channels situated at different context levels. Moreover, this script also defines a set of events, triggering both within the underlying system as well as exclusively within the LARVA monitor. These internal events include timer events denoting that a specified amount of time has elapsed, or channel events notifying that some information has been sent over the channel. It is through these events that the monitoring algorithm used for LARVA knows where to integrate the LARVA monitor with the underlying system.

The LARVA compiler translates the LARVA script into corresponding Java and AspectJ code. AspectJ is a Java implementation for Aspect Oriented Programming techniques (discussed in further detail later). This generated code serves two purposes, these being the implementation for the simulation of DATEs (provided by the generated Java code), as well as the instrumentation of the resulting DATEs with the underlying system (implemented using AspectJ code). It is through this AspectJ code that the resulting DATEs are able to execute actions on the system and have access to system information and events.

Note that LARVA is currently an event-based approach triggering on internal system actions (as opposed to state based triggering on alterations within the system state). Also, the current LARVA implementation offers support exclusively for online runtime verification monitoring, which implies that care has to be taken so as to reduce the system overhead. Keeping this in mind, the remainder of the chapter is dedicated towards the detailed discussion of the notions driving LARVA introduced above.

3.2 DATEs

The following section presents Dynamic Automata with Timers and Events (DATEs), which are the formal backbone driving LARVA, and provide an executable platform through which the defined properties are verified at runtime. We shall formalise three core notions, these being the notions of the event, the symbolic timed automaton and finally DATEs. All definitions shall be presented in an evolutionary manner, starting

from the formalisation of the notion of an event.

LARVA is an event based approach, hence implying that the notion of the event plays a central role in the integration of the runtime verification framework with the underlying system. In fact, DATEs can be considered as communicating symbolic automata with timers, and whose transitions are triggered by events [12]. Three basic events are considered, these being (i) a system event which can fire within the underlying system, (ii) a timer event triggering upon the passage of a set amount of time, and (iii) channel event, used for automata synchronisation and information passing. Moreover, basic events can be used to define composite events defined below

Definition 3.2.1 *Given a set systemevent of events generated within the underlying system, a set of timers timer of type Timer and another set of channels channel of type Channel declared within the monitor, a composite event is defined as*

$$event ::= systemevent \mid (channel?(x : X)) \mid timer@\delta \mid event + event \mid \overline{event}$$

Whereby a composite event is either one of the three possible basic events, a choice between two composite events or a complement of a composite event. Note that the event definition given above slightly differs from that given in [12], more specifically with regards to the definition of the channel event. Whereas DATE channels were previously defined exclusively for synchronisation purposes (a signal carrying no information) alerting any transition listening on that channel, the above definition extends the channel with data passing semantics (passing a value of sort X). Certainly such an alteration should be handled with care, since fundamentally augmenting channels in such fashion may lead to drastic alterations to the results presented within the rest of the DATE framework. However, it turns out that the basic channel signal-based calculus is equally expressive to the full value passing calculus as proven by Milner [30].The essence of this expressive equality lies in the fact that given a channel of sort X in the full calculus, where X is of size n, then the channel in the full calculus can be encoded within the basic calculus using n different signal channels (for example modeling a boolean channel using basic channels would involve two basic channels, one which signals truth and the other which signals falsity). Moreover, this technique can be extended for infinite sorts (such as integers or strings) by using an infinite amount of channels. For a more detailed discussion please refer to [30]. Consequently, we can simply augment the signal channels within the DATE framework to value passing channels without affecting the remaining results within the RV framework.

Definition 3.2.2 *Given a basic event x, and a composite event e, we say that basic event x fires composite event e (written as* $x \models e$*) if (i) x is equal to e, or (ii)* $e = e_1 + e_2$ *and either* $x \models e_1$ *or* $x \models e_2$*, or (iii)* $e = \overline{e_1}$ *and* $x \not\models e_1$*.*

Note that the notion of the firing of composite events can also be extended to work on sets of basic events. Given a set of basic events X and a composite event e, X $\models$ e if (i) x $\in$ X and x $\models$ e, or (ii) e = e_1 + e_2 and either X $\models$ e_1 or X $\models$ e_2, or (iii) e = $\overline{e_1}$ and $\forall\, x \in X \bullet x \not\models e_1$.

With the definition of the event and the corresponding firing of an event complete, we shall now define symbolic timed automata. As the name implies, the structure admits temporal functionality, whereby this functionality is presented through the use of timers. However, symbolic timed automata are more expressive than timed automata [2] since the defined timers allow for timer actions, these being actions which reset, pause or resume the timer [12]. Consequently, a timer can be in either one of two states, running or paused, therefore TimerState = { running, paused }. Also note that the timer assumes a dense time model, whereby time is represented using the real number line.

Definition 3.2.3 *The timer configuration, $\mathcal{CT}$ is a function from timers to the value recorded on the timer and the timer state.*

$$\mathcal{CT} \;::\; Timer \;\rightarrow\; (\,\mathbb{R} \times TimerState\,)$$

Hence, the timer configuration defines the timer state (the current timer value and the associated state) for all the timers defined within the DATE framework. The above timer configuration definition implies that the timer actions $\mathcal{TA}$ are functions from a timer configuration to another ($\mathcal{CT} \rightarrow \mathcal{CT}$), whereby a timer reset sets the timer value to zero, the timer pause sets the timer state to paused, and the timer resume sets the timer state to running.

The following automata are considered symbolic due to the fact that each defined automaton state represent a set of states within the system. If for example we define state *countAboveZero* specifying that a particular count within the system is above zero, this automaton state represents all possible system states where this particular count is above zero. Also, another important point worth mentioning is the fact that the following automata have access to both read and alter the underlying the system state (henceforth represented by the type symbol Θ). This modification usually takes the form of an action on the system, thus represented as ($\Theta \rightarrow \Theta$).

Definition 3.2.4 *A symbolic timed automaton is defined as* $\langle Q, q_0, \rightarrow, B, A \rangle$*, with* Q *states, initial state* $q_0 \in Q$*, transition relation* $\rightarrow$*, bad states* $B \subseteq Q$*, accepting states* $A \subseteq Q$*,* $A \cap B = \emptyset$*. Transition relation* $\rightarrow$ *encapsulates a set of transitions, whereby each transition is defined though (i) the start event (ii) the event expression (defined above) which triggers the transition (iii) a condition on the current system state and timer configuration (*$\Theta \times \mathcal{CT} \rightarrow \mathbb{B}$*) (iv) the timer action executed on the current timer configuration upon traversing the transition (v) a set of channels passing information (of type X) over the channels, (vi) an action on the underlying system (vii) the end state.*

$$Q \times event \times (\Theta \times \mathcal{CT} \rightarrow \mathbb{B}) \times \mathcal{TA} \times 2^{(Channel \rightarrow X)} \times (\Theta \rightarrow \Theta) \times Q$$

The symbolic timed automaton defined above are slightly altered from that defined in [12], since the above definition caters for the augmented notion of value passing channels whereby information can be sent over these channels. Note that when an automaton stumbles on an accepting state, it is not allowed to traverse any more transitions since an accepting state implies that the property has been satisfied.

The above symbolic timed automaton assumes a total ordering $<$ exists both at a per transition as well as at a per automaton basis. This implies that given a set of transitions within the same automaton which trigger on the same event expression, we avoid non-determinism by applying this ordering so as to determine the order with which transitions are executed. This logic is analogously applied to a vector of automata, whereby given transitions defined within distinct automata which simultaneously trigger, the transition execution ordering is dictated by the ordering defined on the automata. Consequently, upon the firing of a set of events, the evaluation algorithm (i) chooses the highest priority transition dictated by both orderings (firstly choosing the highest priority automaton, then choosing the highest priority transition within the automaton which has triggered on the events) (ii) executing the transition, hence also executing any necessary timer actions, sending information down channels and also actions on the system, and (iii) repeating this process until no triggered transitions are left.

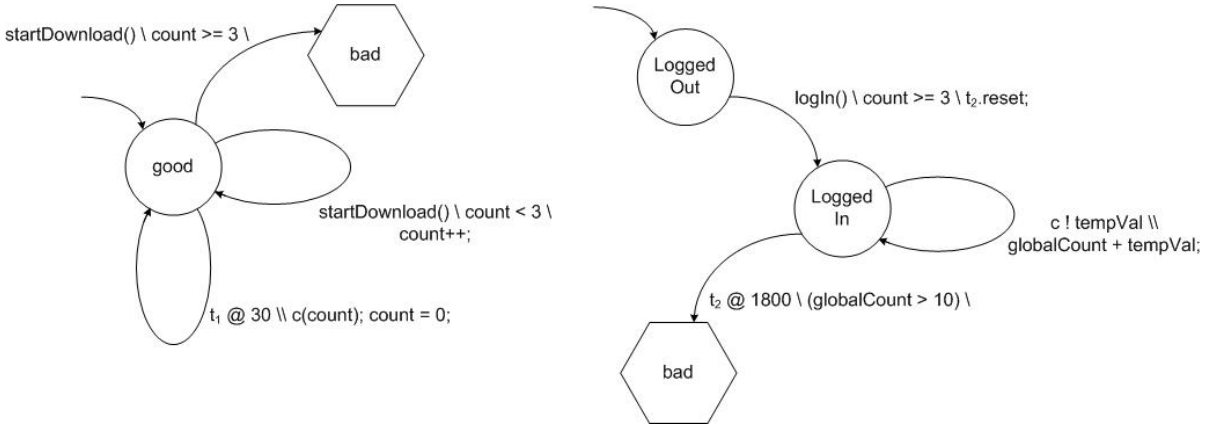
Definition 3.2.5 *A symbolic timed automaton M performs a full step from the set of scheduled system events X, system state* $\theta \in \Theta$*, timer configuration* $T \in \mathcal{CT}$ *and state* $q \in Q$ *(the current state), to* X'*,* Q' *and* q' *using timer update* t' *if* $(X, \theta, q) \Rightarrow^{T}_{t'} (X', \theta', q')$ *given that* $q \notin A$ *and* (q, e, c, t', O, f, q') *is the largest transition (as defined by* $<$*) in* $\rightarrow$ *such that (i)* $X \models e$*, (ii)* $c(\theta, T)$ *holds, (iii)* $\theta' = f(\theta)$*, (iv)* $X' = O$*.*

If no such transition exists implying that that the fired events do not trigger any transition, we write $(X, \theta, q) \Rightarrow^{T}_{\mathrm{id}} (\emptyset, \theta, q)$. Note that the notion of an automaton full step can be extended to a vector of automata as defined below

Definition 3.2.6 *Given a vector of n automata* $M = \langle M_1, M_2, ..., M_n \rangle$ *in states* $q = \langle q_1, q_2, ..., q_n \rangle$ *all of which are sharing the same set of timers in timer configuration* T*, we say that* $(X, \theta_0, q) \Rightarrow_t^T$*,* (X, θ_n, q') *if (i) for each* $1 \leq i \leq n$*,* $(X, \theta_{i-1}, q_i) \Rightarrow_{t_i'}^{T_i'}$ (X_i', θ_i, q_i')*, (ii)* $t' = t_n' \circ t_{n-1}' \circ ... \circ t_1'$ *(iii)* $X' = X_1' \cup X_2' \circ ... X_n'$*, (iv)* $T_i' = (t_{i-1}' \circ t_{i-2}' \circ ... \circ t_1')(T)$*, (v)* $T' = t_n'(T_n')$*, and (vi)* $q' = \langle q_1', q_2', ..., q_n' \rangle$

Where the triggering of automata is determined by the assumed $<$ ordering previously discussed. In practice, the full step over a vector of automata is useful since multiple properties are usually defined within a typical script, all of which are listening to the same set of events. Note that the timer actions are accumulated so that all conditions are evaluated on the same initial timestamps [12].

The following symbolic timed automata define a system property specifying limitations on user download behaviour in order not to put the system under too much load. This limitation specifies that any user should not download more than ten files for the first thirty minutes since login, while also disallowing the starting of three user downloads within any thirty second time period. Note that the following example assumes variables count and globalCount, clocks t_1 and t_2, and channel c.



The leftmost automaton is responsible for ensuring that no more than three downloads occur within a thirty second period. Whereas one transition is responsible for counting the number of downloads started, another transition triggers upon the elapsing of thirty seconds, sending the current count of channel c and resetting the timer. If the count happens to exceed three before the count reset, the property enters a bad state. The rightmost automaton ensures that the amount of downloads after user login should not exceed ten. Upon login t_2 is reset. Another transition is listening to channel c, adding the value sent over the channel to the global count. If this global count exceeds ten upon the elapsing of thirty minutes, the property enters a bad state.

With the notion of symbolic timed automata defined, the next step is to augment this notion with dynamic behaviour, capable of runtime creation of new automaton instances triggered by the system behaviour during the system execution. The augmentation of symbolic timed automata with dynamic behaviour are known as Dynamic Automata with Timers and Events (DATEs), and admit a wealth of practical applications. Such dynamic nature could for allow for verification on a per object basis, since we could generate a new automaton upon the creation of a new object (representing the context). If for example we wish to verify that each user is assigned an session number upon login, we could verify such a property on each session object created within the system. This idea of context can also be applied to above symbolic timed automaton example, whereby we would like to check the above property on a per user basis. Note that each context can in general define a local set of variables, clocks and channels. Consequently, we shall henceforth assume that such constructs are embedded within the underlying system.

Definition 3.2.7 *A DATE $\mathcal{M}$ is a pair (M,v) consisting of (i) an initial set of symbolic timed automata M, and (ii) a set of automaton constructors v. An automaton constructor is the construct responsible for the dynamic creation of automata at runtime and is defined as a triple consisting of (i) an event expression, (ii) a boolean condition on the current system state and timer configuration, and (iii) a function which generates a new automaton instance based on the current system state and timer configuration. Thus, automaton constructors v are of the form*

$$event \times (\Theta \times \mathcal{CT} \rightarrow \mathbb{B}) \times (\Theta \times \mathcal{CT} \rightarrow Automaton)$$

Definition 3.2.8 *Given time configuration T, the set of scheduled system events X, current system state θ, the corresponding triggered automata (written tr(T,X,θ)) is defined as*

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

Since certain DATEs are defined to have a set of local variables, clocks and channels, we shall assume that upon the triggering of an automaton constructor a fresh copy of each shall be created and stored within the underlying state.

Definition 3.2.9 *The configuration of a DATE of type DATEConf is defined as a triple containing (i) the timer configuration, (ii) the state of the underlying system, and (iii) the state of the set of currently running automata, thus*

$$DATEConf :: \mathcal{CT} \times \Theta \times 2^{Q}$$

Definition 3.2.10 *A DATE performs a full step from DATE configuration (T, θ, q) to (T', θ', q') which is triggered by the set of system actions X (written (T, θ, q) $\Rightarrow$ (T', θ', q')) if for some n,*

$$(X_0, \theta_0, q_0) \Rightarrow^{X'_1}_{t_1} (X_1, \theta_1, q_1) \Rightarrow^{X'_2}_{t_2} \ldots \Rightarrow^{X'_n}_{t_n} (X_n, \theta_n, q_n) \Rightarrow^{X'_{n+1}}_{t_{n+1}} (\emptyset, \theta_{n+1}, q_{n+1})$$

such that (i) $X = X_0$, $q_0 = q$, $\theta = \theta_0$ *and* $\theta' = \theta_{n+1}$ *(ii) the final timer configuration is updated according the the accumulated timer actions (for* $1 \leq i \leq n+1$*)* $T'_i = (t_{i-1} \circ t_{i-2} \circ \ldots \circ t_1)(T)$*, (iii)* $T' = t_{n+1}(T'_{n+1})$*, and (iv) the automaton states are updated by DATE triggers* $q_i = q_{i-1} \oplus \bigcup_j q_{0_j}$ *where* $\bigcup_j q_{0_j}$ *is the set of all initial states defined for all automata in* $tr(T'_i, X_i, \theta_i)$

A DATE full step is called a *good full step* if no bad states appear within the state vectors. It is apparent that the above definition of automata channels may give rise to situations where an automaton triggers indefinitely. Hence, in order to avoid this issue of livelock, [12] defines a restriction on the automaton structure as defined below

Definition 3.2.11 *Given automaton M, we define out(M) to be the union of all output channels used by the transitions in M, and in(M) to be the union of all channels which transitions in M listen to. Consequently, the channel dependency relation for M (written as dep(M)) is defined as in(M)* $\times$ *out(M). In order to avoid the issue of livelock, we require automata to be loop free. Hence, given a DATE* $\mathcal{D}$ *containing n automata,* $\mathcal{D}$ *is said to be loop free if* $\forall\, c : Channel \bullet (c, c) \notin (\bigcup_i dep(M_i))^+$.

Intuitively the above definition specifies that if a channel is used by an automaton, it is exclusively used for outputting or inputting a value. Only loop free automata are henceforth capable of performing a full step. Finally, note that any automaton which has stumbled on an accepting state can be removed from the DATE structure (along with its associated local variables, clocks and channels), since by stumbling on an accepting state the property represented by the automaton has been satisfied and hence inherently can not traverse another transition.

3.3 The LARVA Script

With the DATE mathematical framework defined, the next step is to specify a language with which to express DATE constructs acting as system properties, which the underlying system behaviour is required to abide to. This language is defined in the form of the LARVA script presented below. In a similar approach to the presentation of DATEs, LARVA shall be presented in an evolutionary manner through the discussion of the script BNF defined for various script sections, as well as through examples. Recall that LARVA is a runtime verification tool for Java programs, which implies that LARVA embeds Java code within the script where required (more below).

States

Since DATEs define three distinct state types, these being accepting, normal and bad states, LARVA defines three distinct blocks for the definition of states. Moreover, each declared state can be associated with a code block which is executed upon entering the state. The following BNF represents the LARVA script BNF for the definition of states.

StateName	::=	identifier
EventName	::=	identifier
Condition	::=	BooleanExp
Action	::=	JavaCode
Transition	::=	StateName '→' StateName '[' EventName '\' Condition '\' Action ']'
TransitionList	::=	Transition | Transition TransitionList
TransitionBlock	::=	'*TRANSITIONS*' '{' TransitionList '}'

Code associated within a state can be used for initialisation or reparatory action, as exemplified below

```
STATES  {
    BAD  { loginCountExceeded { blacklistIp(); }  }
    NORMAL  { userLoggedIn  { downloadCount = 0;  }  }
    STARTING  {  userLoggedOut  }
}
```

Transitions

Inherent in the notion of the symbolic timed automaton is the concept of the transition. A transition triggers the automaton to change state upon the occurrence of an event and the satisfaction of a condition. Moreover, an action is executed upon the traversal of a transition. Hence, the LARVA BNF for the definition of transitions is defined as

StateName	::=	identifier
EventName	::=	identifier
Condition	::=	BooleanExp
Action	::=	JavaCode
Transition	::=	StateName '→' StateName '[' EventName '\' Condition '\' Action ']'
TransitionList	::=	Transition | Transition TransitionList
TransitionBlock	::=	'*TRANSITIONS*' '{' TransitionList '}'

Hence, a transition is of the general form startState -> endState [eventExpression \ condition \ action]. Transitions are defined within one section as exemplified below

```
TRANSITIONS  {
    userLoggedOut -> userLoggedIn [ LogIn \ count >= 3 \ globalCount = 0;]
    userLoggedIn -> userLoggedIn [ startDownload \ count < 3 \ count++;]
    userLoggedIn -> userLoggedOut [ startDownload \ count >= 3 \ logout();]
}
```

Note that the transition definition order represents the transition ordering assumed on transitions discussed in the previous section. Hence, if more than one transition triggers simultaneously (such as the last two defined transitions which trigger on the event startDownload), the execution order is defined by the order the transitions are defined. Also note that both the condition as well as the action is defined using Java syntax. Hence, the action component may contain a block of Java code.

Properties

Automata are essentially defined through states and transitions. Hence, given that defined automata represent properties which the system behaviour is to adhere to, LARVA defines properties through a state, transition pair as exemplified below.

```
PROPERTY loginMonitor  {
  STATES  { ... }
  TRANSITIONS { ... }
}
```

Hence, using the previous state and transition BNF declarations, the BNF for the definition of a LARVA property is defined as

PropertyName ::= identifier
PropertyBlock ::= '*PROPERTY*' PropertyName '{' StateBlock TransitionBlock '}'

Events

LARVA requires the definition of two core aspects within the script, these being the property declarations and events which act as the link between the properties and the underlying system. Events are declared within the script and consequently listened to by transitions defined above, whereby such events take the form of method calls or exception throws within the system. Note that LARVA opts to avoid monitoring variables since doing so would potentially give rise to substantial overheads given that variables valuations may change rapidly. In truth this issue is mitigated since the object oriented principle of encapsulation requires access to such variables to be defined through the method getter and setter, which are method calls.

The event declarations specified within LARVA go beyond the simple capturing of method calls within the underlying system. In fact, events can also be used to extract context information, which primarily takes the form of (i) method parameters, (ii) the target object which the method is executed on, (iii) the object returned by the method call, (iii) upon a method throwing an exception, and (iv) upon a method handling an exception (the execution of a catch block) . Such information is often vital when property declarations would like to differentiate between events. If for example we would like to

distinguish between a successful and unsuccessful login event, we would require the the analysis of the returned object containing the login attempt result. Also note that each event can be declared with a "call" or "execution" tag. An event capturing a method call captures the event from the point of view of view of the code which makes the call, whereas capturing a method execution captures the event from the executing event point of view. Although the differences in both are slight, it is sometimes required especially when the executing method call source code is not available, or the method call itself is executing by using Java reflection. The following are a few sample event declarations as defined in LARVA

```
EVENTS  {
    downloadStarting(User u, String fileName) = {call u.eventDownloadFile(fileName)}
    attemptedLoginEvent(String result) = {execution User u.PASS(String password)
                                                          uponReturning (result) }
    ...
}
```

Events are declared in an EVENTS block prior to any property declaration. Note how the first event declaration extracts the user object as well as the file name downloaded by capturing the start of download event. Moreover, note how the second transition extracts the login attempt result by using the "uponReturning" tag. This extracted information is then available to transitions listening on the event, whereby such information can then be referred to both in the transition condition as well as the associated action. It is important to note that LARVA enables for the distinction of overloaded methods the method call declaration by analysing the amount of defined parameters.

In general, all information extracted from the system is to be declared with an appropriate variable name and type as exemplified above (such as "String fileName"). Although such variable declarations can be placed throughout the event declaration (hence both within the even declaration as well as within the event parameters), information which we want to make public to the rest of the transition (listening on the event) is to be passed through the event parameters, whereas information which we do not require further is to be left out (from the event parameter declarations). Note that since multiple events may be declared all with their local event parameters, in order to avoid ambiguity we require that event parameters for different events may be given the same names, as long as they have the same types. Hence, given the above declaration of event downloadStarting, we can declare further events declaring an event parameter fileName, however any such declaration must ensure that fileName is a string event parameter.

Another interesting feature offering further flexibility for the definition of events is the declaration of events using wildcards. This enables the user to declare events such as all method calls on a particular object, or all calls to a particular method call regardless of the target object, as exemplified below

```
EVENTS  {
    userAction(User u) = {call u.*()}
    loginAttempt() = {execution *.PASS() }
    ...
}
```

This notion of wildcards is in fact extended to any declaration location within the event. Hence, a wildcard can be placed instead of an object type, name as well as instead of method parameters, leaving them unbound. This is especially useful when specifying an overloaded method declaration, whereby we require the specification of a specific overloaded version without requiring the information defined by all parameters, such as

```
readByte(InputStream in) = {call in.read(byte b,*,*)}
```

As to summarise, an event declaration can take one of the following general templates, or one multiple combinations (of these templates).

- eventName(Type1 arg1) = methodCall(Type1 arg1, Type2 arg2)
- eventName(Type1 target) = Type1 target.methodCall(...)
- eventName(Type1 return) = methodCall(...) uponReturning (return)
- eventName(Type1 ex) = methodCall(...) uponThrowing (ex)
- eventName(Type1 ex) = methodCall(...) uponHandling (ex)

Note that in the last two cases yet to be discussed, the event declaration captures the throwing or handling of an exception executed by a method. In both cases, we can extract the exception object which was thrown/handled.

LARVA identifies the issue that although the presented logic for the description of events is flexible, it is sometimes not sufficient with regards to the issue of information extraction. As an example scenario consider the situation where we would like to know the time an event occurred, whereby such information is not directly extractable from the method call. Hence, LARVA defines the where clause, which allows for the extraction of information not directly associated or extracted from the event. Hence, an additional where clause can be added to the end of each event declaration, as exemplified below extracting the time when the download was started

```
downloadStarting(User u, String fileName, long time) =
                                    {call u.eventDownloadFile(fileName)}
                                       where {time = System.currentTimeMillis();  }
```

Event Collections

LARVA allows for the definition of event collections. Such collections are ideal for situations where we would like an event to trigger upon the triggering of another given a set of possible events. Given the scenario where the system implementation defines three methods, methodA(), methodB() and methodC(), we would like to define an event which triggers if any other event but the execution of methodA. This example is specified below

```
EVENTS  {
    eventA() = {...}
    eventB() = {...}
    eventC() = {...}
    eventNotA() = { eventB | eventC }
    eventNotB() = { eventA | eventC }
    eventNotC() = { eventA | eventB }
    ...
}
```

Event collections are separated by the | symbol, and separates either events referred to by their name (hence previously defined within the event block), or a whole event declaration using the previous event declaration syntax. This implies that through the use of event collections, we can specify nested events, such as

```
eventCollection() = { {methodA()} where {...} | eventB }
```

A direct result of the augmented syntax for the declaration of nested events is the effect on parameter and where clause declarations. In general, an event collection can also define a set of parameters, and it is the responsibility of each sub event declaration to set all the parameter valuations, as exemplified below

```
eventCollection(Type1 arg1, Type2 arg2) = { {Type1 arg1.methodA()} where {arg2 = 2;}
                                                                    | eventB(arg1,arg2) }
```

Hence, event parameters declared by an event collection can be set either directly by the sub event itself, or declared within the where clause (in case of arguments which have not been directly set by the sub event).

As exemplified above the where clause can either be used within a sub event declaration, as well as for the event collection declaration. Note that in the case of the event collection where clause declaration, this particular where clause is applied to all sub events which might fire within the collection.

```
eventCollection(Type1 arg1, Type2 arg2, Type3 arg3) =
                  { {Type1 arg1.methodA()} where {arg2 = 2;} | eventB(arg1,arg2) }
                                                                    where {arg3 = 0;}
```

This implies that if the event fired by the execution of methodA or eventB fires, the event collection fires too, with arg3 being set to 0 in either case. Such "global" where clause declarations are useful for reducing the amount of where clause declarations defined for each sub event. In the case of conflicts arising from where clause declarations where event parameters are declared more than once, as shown below

PropertyName ::= identifier
PropertyBlock ::= '*PROPERTY*' PropertyName '{' StateBlock TransitionBlock '}'

Note that the innermost valuation is never overridden, thus implying that in the case of eventCollection firing due to the firing of the sub event declared for methodA, the valuation for arg3 is 2 (and not 0 as specified by the outermost where clause).

One final note regarding the declaration of events is the fact that we can define an unlimited amount of nested event collections, all of which abide by the event parameter and where clause rules defined above.

Event BNF

The following specifies the complete syntax BNF for the declaration of events as well as event collections.

Type ::= identifier
VariableDeclaration ::= '$*$' | identifier | Type identifier
MethodName ::= identifier
EventName ::= identifier
ArgumentList ::= VariableDeclaration | VariableDeclaration ArgumentList | ϵ
EventVariation ::= 'uponReturning' '(' VariableDeclaration ')'
| 'uponThrowing' '(' VariableDeclaration ')'
| 'uponHandling' '(' VariableDeclaration ')'
MethodDeclaration ::= VariableDeclaration '.' MethodName '(' ArgumentList ')'
PrimitiveEvent ::= MethodDeclaration | MethodDeclaration EventVariation
EventList ::= PrimitiveEvent | CompoundEvent | EventName
| EventName '|' EventList
CompoundEvent ::= '{' EventList '}' | '{' EventList '}' where '{' statements '}'
Event ::= EventName (ArgumentList) '=' CompoundEvent
Events ::= Event Events | ϵ
EventsBlock ::= '*EVENTS*' '{' Events '}'

Variables

Apart from the declaration of properties and events, LARVA also allows for the declaration of additional variables which can be referred to by property declarations. This allows for the straightforward declaration of temporary information (for example the current logged in user ip), and also as a counter (storing the number of consecutive bad

logins). All variable declarations follow the Java declaration syntax, and can declare an instance of any required object, as exemplified below

```
VARIABLES  {
    int counter = 0;
    String userName = "";
}
```

Note that in the case that we require the instantiation of a user defined (or non-standard) Java class implementation, this is also allowed within the VARIABLES section as long as the require packages are imported (more below). Also note that, as will be specified below, the variables section shall be used for the declaration of clocks and channels. Below is the BNF for the declaration of the variables block.

Type	::=	identifier
VariableName	::=	identifier
VariableDecl	::=	Type VariableName ';'
		| Type VariableName '=' JavaStatement ';'
VariableList	::=	VariableList VariableDecl | ϵ
VariablesBlock	::=	'*VARIABLES*' '{' VariableList '}'

Clocks

Recall that DATEs define real time functionality through the use of timers/clocks. Hence, we require a mechanism within the LARVA script which allows for (i) the definition of clocks, and (iii) the execution of actions on these clocks within Java code. Also recall from the previous chapter that clocks trigger events upon the elapsing of a specified amount of time. Hence, we also require to slightly alter the above definition of events for the specification of clock events. The declaration of clocks simply involves the declaration of a clock within the VARIABLES section, whereby the declaration is analogous to any other variable declaration.

```
VARIABLES  {
    Clock clock;
}
```

Any LARVA script can declare an unbounded amount of clocks, all of which can be used to define multiple clock events. Note that whereas older LARVA implementations required a separate thread for each clock, thus making clocks an expensive resource, newer LARVA implementations simulate all clocks on one thread. Hence, one can use as many clocks as required without posing too much of an additional overhead. A clock event entails the declaration of an amount of time which triggers an event upon the elapsing of this time amount.

```
clockEvent = {c @ n}
```

Where n is a floating point number. Note that whereas the default time unit is the second, LARVA allows for the definition of floating point numbers, thus also allowing for the declaration of millisecond durations. An additional syntactic sugar provided is the "c @% n" clock event, which triggers repeatedly after every n time units (unlike the previous clock event declaration which triggers as a one off event). The BNF for the declaration of a clock event is the following.

number	::=	positive integer
ClockEvent	::=	ClockName '@' number | ClockName '@%' number
PrimitiveEvents	::=	PrimitiveEvent | ClockEvent
EventList	::=	PrimitiveEvents | CompoundEvent | EventName
		| EventName '|' EventList

The remaining issue is the declaration of actions on clocks. Note that a clock declaration is treated as any other object declaration, which implies that any action on clocks can be implemented through methods, and such actions can be executed as method calls within any Java code block. The following methods are defined for the clocks

- reset() — resets the internal clock value. Note that the clock valuation starts running as soon as it is created.
- current() — returns a double specifying the current internal clock value.
- compareTo() — accepts a double and returns an integer. If the integer is zero, the current clock value is equal to the parameter value. If the result is positive, the clock value is larger than the parameter. Else, the clock value is smaller than the parameter.
- off() — switches off the clock, resetting the current statistic valuation in the process.
- pause() — pauses the clock, thus freezing the internal clock value. Pausing an already paused clock has no effect.
- resume() — resumes a paused clock, thus allowing the internal value to increase once more. Resuming a running clock has no effect.

Thus method calls such as "clock.reset()" resets the clock, whereas "clock.current()" returns a double specifying the internal clock value, and can be embedded within any Java code block.

Channels

Automata within the DATE framework are able to communicate through the use of channels. Such communication can be used for simple synchronisation purposes (change

an automaton state upon another automaton reaching another state), as well as for information passing requirements (pass the value which broke a property from one automaton the another). Note that channels are of broadcast channels, implying that all transition listening to a channel are notified upon the passing of information on that channel. Channels are declared in an analogous fashion to clocks, within the VARIABLES section as exemplified below

```
VARIABLES  {
    Channel channel;
}
```

The notion of the channel allows for two actions on channels, these being the sending of information and the retrieval of information from a channel. The sending of information is executed by the send() method, whereby this method is overloaded defining a no parameter as well as a parameter version (accepting an object of type Object). The no parameter version is used for synchronisation purposes, whereas the parameter version is used for the sending of information over the channel, as exemplified below

```
channel.send(2);
```

Thus the above code sends the value 2 over the channel. Note that the above method can be embedded within any Java code block. The retrieval of information is specified through a channel event, as exemplified below

```
EVENTS  {
    receivedChannelValue(Object o) = {channel.receive(o)}
}
```

Thus any transition listening to the above channel event is triggered each time a value is sent over the channel, and also has available the object passed over the channel, as shown below

```
TRANSITIONS  {
    startState -> normalState [receivedChannelValue \ \ System.out.println(o);]
}
```

Context

The remaining core issue yet to be fully characterised within LARVA is the issue of context. Currently, all defined constructs so far can be put in the global context, which essentially contains a variables block, and events block, and a list of properties.

VariablesBlock	::=	'*VARIABLES*' '{' *VariableList* '}'
PropertyBlocks	::=	PropertyBlock PropertyBlocks | ϵ
Context	::=	VariablesBlock EventsBlock PropertyBlocks
GlobalContext	::=	'*GLOBAL*' '{' Context '}'

However, as previously motivated we require the definition of contexts linked to objects, such that a new context is created upon the creation of a new object. This context declaration through the FOREACH construct, whereby FOREACH (Session s) creates a new context upon the creation of each new session object. Each context is to contain a variables block, and events block, a list of properties and a list of subcontexts. However, the properties, contexts, events etc are local to that context, and are hence available only to that context (and hence also to sub contexts). It is crucial to note that each event declared within a subcontext is to define its context through the where where clause, as exemplified below

```
FOREACH (Session s)  {
    VARIABLES { ... }

    EVENTS {
        sessionCreated() = {*.createSession() uponReturning (Session session)}
                                                                where { s = session; }
    }

    PROPERTY { ... }
    ...
    FOREACH ( ... ) { ... }
    ...
}
```

LARVA allows for nested contexts, such as properties *foreach* session *foreach* connection. In fact, thr resulting set of possible automata is the cross product resulting from all the session and all connection instances [12]. For example, we would like to monitor connections (used for downloading purposes) within each session such that if a connection is idle for a specific amount of time it is automatically terminated, as exemplified below

```
FOREACH (Session s)  {
    ...
    FOREACH (Connection c)  {

        VARIABLES { ... }

        EVENTS {
            connectionCreated() =
                 {Session session.openConnection() uponReturning (Connection conn)}
                                                    where { c = conn; s = session; }
        }

        PROPERTY monitorConnectionIdle{ ... }
    }
}
```

Note how the event connectionCreated declared within the Connection context declares the whole context i.e. both the context for the connection object as well as the

context for the Session object. In general, an event declared within the i^{th} subcontext must declare all i subcontexts. The following is the defined BNF for the inclusion of subcontexts within the global context

Type	::=	identifier
VariableName	::=	identifier
Context	::=	VariablesBlock EventsBlock PropertyBlocks SubContexts
SubContext	::=	'*FOREACH*' '(' Type VariableName ')' '{' Context '}'
SubContexts	::=	SubContexts SubContext | ϵ
GlobalContext	::=	'*GLOBAL*' '{' Context '}'

The creation of the notion of context gives rise to the requirement of context rules which govern variable and event access across contexts. The context rules adopted are analogous to those defined for any standard programming language, where any subcontext can refer to variables and listen to events declared within its super contexts. However, super contexts can not refer to variables and events declared within its sub contexts. In the case of a constructs listening to events declared within a super context, note that this event acts as a broadcast event to all construct instances within the subcontext, since this event does not have enough context to distinguish between subcontexts. Using the previous example, had property monitorConnectionIdle (which is instantiated multiple times, one for each connection) is listening to an event declared within the session context, then this event will trigger all instances of monitorConnectionIdle within the same Session context. Moreover, if this property listens to an event declared within the global context, this event triggers all instances of monitorConnectionIdle across all Session contexts, since this particular event declaration (being declared within the global context) does not have enough context information to distinguish between Session contexts, let alone Connection contexts.

One final issue worth mentioning is the effect of context on clocks and channels. Whereas clocks are contextual, and are hence exempt from the requirement of the declaration of context within a clock event (hence, clocks and their corresponding events can be declared within a context without requiring the definition of context), channels are broadcast in nature. Hence, although channels and their corresponding channel events can be declared within a context, these events still require the definition of the corresponding context. Alternatively, all channels can be declared within the global context and listened to by the sub contexts as required. An interesting technique (used later in the definition of our statistical framework) which solves the issue of context with respect to channels is to encapsulate the information which we wish to send over the channel within an object which tags the information with the corresponding context. Hence, by sending this object over the channel not only are we sending the actual information, but also the associated context.

Invariants

Invariants are values within the underlying system which we require to remain unchanged during the system execution. We could for example require that system ID remains unchanged, or that the system operating system kernel process priority remains the highest throughout the operating system's execution. Such invariants are declared as follows

```
INVARIANTS  {
    int sysID = System.getSystemID();
}
```

The following BNF extends the notion of context to include an additional block which allows for the declaration of invariants.

Type	::=	identifier
VariableName	::=	identifier
VariableDecl	::=	Type VariableName ';'
		| Type VariableName '=' JavaStatement ';'
VariableList	::=	VariableList VariableDecl | ϵ
VariablesBlock	::=	'*VARIABLES*' '{' VariableList '}'

Imports & Methods

Any LARVA script allows for the specification of an imports block as well as a methods block. As the name implies, the imports section is responsible for importing of any additional Java packages, as exemplified below

```
IMPORTS  {
    import Java.io.*;
    import java.sql.*;
    import Java.util.*;
    ...
}
```

The IMPORTS section is to be the very first block defined within the LARVA script. As can be seen, the actual syntax for the importing of packages is identical to that defined for Java. Using such importing features, we can implement complex packages and functionality and simply import the code to the LARVA script. Hence, any given LARVA script can easily specify complex logic as required. Moreover, any LARVA script allows for the definition of methods at the bottom of the script through the declaration of a METHODS block. These methods can then be referred to as required within any Java code block defined within the script (such as transition actions or code executed upon entering a state).

```
METHODS {
    public class SC{
```

```
        double static methodA ( int num ) { ... }

        String static methodB ( double d ) { ... }
    }
}
```

All method declarations are standard Java methods. Also, these declared methods can refer to functionality imported within the IMPORTS section. Note that all declared methods are declared within a class placeholder (called SC). Hence, any method declaration within a code block is to be written with an "SC." prefix. Hence, if we would like to for example call methodA, this would be written as "SC.methodA(...)". Finally, note that all declared methods need not be declared with access specifiers such as private or public. However, it is required that all methods are declared static.

3.4 Conclusions

This chapter introduced the notions of the DATE mathematical framework, as well as the LARVA script capable of expressing DATEs for the use of online runtime verification. DATEs are an automaton-based logic with additional real-time functionality, automaton communication capabilities, as well as allowing for the expression of contextual properties. Given the rich expressivity afforded to the framework (more so than most other available runtime verification tools), as well as the intuitive nature it is presented makes the choice of DATEs as the underlying mathematical framework ideal for our requirements. This choice is compounded by the LARVA script design, since its design is highly modular, easy to use and understandable making it highly extendible. In truth, the choice of LARVA was also dictated by availability factors, given that this is the only tool which we have access to the its source. Consequently, we envisage the extension of this presented tool and logic to be a fruitful task.

4. Extending Runtime Verification with Statistics

The following chapter primarily serves two purposes, these being the motivation for the augmentation of runtime verification of statistics, and also the presentation of a comprehensive overview of current approaches to the field. Moreover, following this we also present an analysis of the current approaches while keeping in mind the choice of LARVA as the underlying runtime verification framework.

The following section gives an overview of the augmentation of runtime verification on statistics, primarily focusing on the motivation of why such an integration is fruitful explained through numerous examples. This is succeeded by a discussion regarding the issue of online statistical computation. Following this, a comprehensive taxonomy of current approaches to the field are presented, starting from the discussion of a theoretical framework for the collection of statistics over runtime executions, and is followed by the discussion of two runtime verification tools (the only two encountered) which are sufficiently expressive for the specification of statistical properties. Finally, a discussion of current approaches is presented keeping in mind the broad aims and requirements identified of a statistical framework (discussed previously).

4.1 Overview

The notion of collecting statistics over runtime executions can be seen as an augmentation of runtime verification concepts. Whereas a runtime verification property is said to *check* for adherence of an event trace to a specification (hence returning a boolean result; either truth if adhered to or failure otherwise), statistics are *evaluated* over the current execution (returning a result of the appropriate type, usually an integer or real number) [17]. We identify a set of core issues which make the augmentation of runtime verification with statistical capabilities a very advantageous endeavor, and are presented below.

Firstly, it is often better to watch for an indicator of failure, rather than to wait for the actual specification violation [17]. The specification "The application should not contain memory leaks" is of little use, since by the time the specification is broken the program would have crashed. This specification can be reworded into "The count of memory allocations and de-allocations should be equal", whereby through the use of counting (a statistical measure) we can capture a situation leading to an impending memory leak, hence taking the necessary reparatory action accordingly a priori. Another example is related to packet transmission within a network. Suppose the requirement for the enforcement of the property "In case of failure, packet retransmission should not go on indefinitely". Using traditional logics used in runtime verification the expression of such a specification is awkward, since they lack the functionality for quantifying the notion of indefinite transmission. However, such a property can be slightly altered into "In case of failure, packet retransmission should not exceed fifty retrials". Obviously the number fifty can be altered as deemed fit. However, what is important is the fact that using statistical properties the above examples can be specified in a straightforward manner.

System requirements can be broadly dissected under two categories, these being functional and non-functional requirements. Whereas a functional requirement usually specifies what the system should do, non-functional requirements are inclined to specify internal system qualities, usually specifying how the system should be. Requirements from both categories are often used in practice, with functional requirements specifying specific behaviour, while non-functional requirements are usually used as an arbitrator of the system operation. What is remarkable is the fact that although most logics used in runtime verification are very adept and focused on specifying functional requirements, there is an apparent lack of logics available for the specification of non-functional requirements. This gap is filled through the augmentation of statistics over runtime verification, whereby statistics collection is used to quantify the internal system state. Hence, the specification of requirements relating to performance, security, and quality of service can all be expressed using statistics. If the developer of a peer to peer network requires the network throughput to be constantly above a certain set level in order to deem the transfer rate acceptable, we can encode this requirement through the analysis of the packet retransmission rate.

From a utilitarian point of view, statistics collection at runtime admits multiple applications in various fields. Such fields include user modeling and data mining, whereby if for example a user visits a cookery website for a statistically significant amount of times, then s/he probably is interested in cooking. Another field benefitting from online statistics collection is intrusion detection, mostly with intrusion detection using statistical anomaly detection techniques. If user behaviour is consistently low risk and admits a statistically low variance (hence approximately adhering to the same pattern behaviour), only to suddenly start intensively accessing sensitive resources is a big indication of in-

trusion. Performance profiling and related measures for quality of service are other fields which benefit from such work, whereby measures for network throughput or percentage delay is expressible using properties of a statistical nature. Finally, testing and program profiling are other applicable fields, whereby statistics can help discriminate important test cases from the less useful [17], while program profiling can make use of statistics for the identification of memory leaks, percentage time spent in certain function calls, as well as issues related to path coverage.

A final issue worth mentioning is the fact that statistics may also increase the expressiveness of the underlying logic (used for the expression of runtime verification properties), as well as the conciseness of resulting specifications. As explained in [14], statistical information enables the expression of context free properties by modeling stacks. Given a two alphabet stack (0 and 1), each pop can be implemented as a division by two (thus removing the least significant bit), while push can be implemented by a multiplication by 2 followed by addition (thus setting the least significant bit). The expressive power of context free properties enables the specification of useful properties such as "every request has a matching grant" or "every start of transfer must have a corresponding end". In fact, an example logic which benefits in terms of expressivity is the logic defined for LOLA [14], whereby the authors theorise that LOLA specifications using only boolean streams cannot express context free grammars (more below). On the other hand, adding statistical capabilities to a logic which is already sufficiently expressive to specify context free grammars will not result in an increased logic expressivity. Nevertheless, this is not to say that property specification within this logic cannot benefit in terms of conciseness, as exemplified below

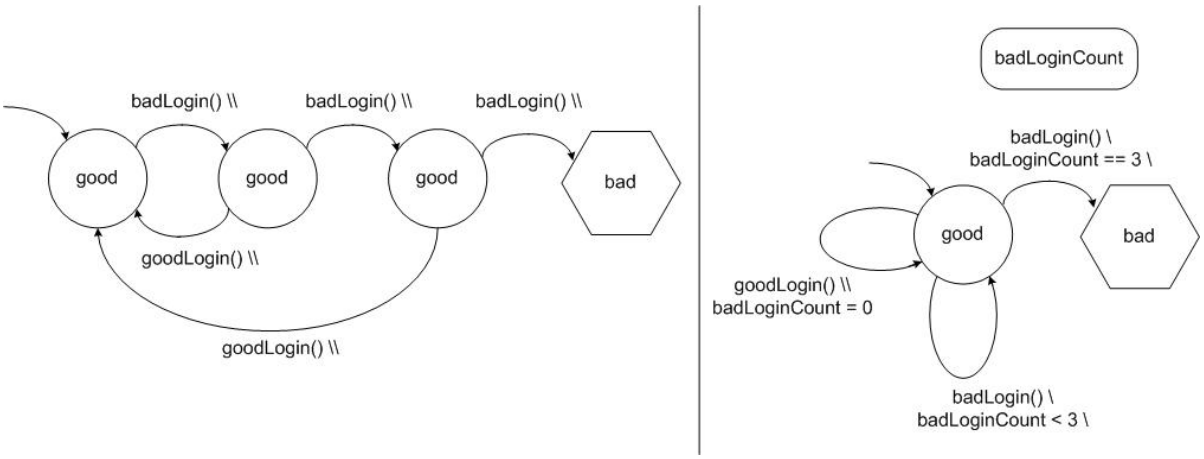


Figure 4.1: (a) An example specification monitoring bad logins. (b) The same specification rendered more concise through statistic badLoginCount.

Leftmost is a DATE specification declaring that the number of consecutive bad logins

never exceeds three, whereas rightmost we specify the same property through the use of statistic badLoginCount, which counts the number of bad logins. Note how badLoginCount is reset upon a good login, such that the statistic is in fact rendered to counting consecutive bad logins. Although the example is trivial, note how a four state, five transition property is converted to a two state, three transition semantically equivalent property simply by using a statistic. It is envisaged that property conciseness is even more substantial for more complex properties.

Through the highlighting of such issues it becomes clear that augmenting statistics with runtime verification is indeed worthwhile. Not only is the collection of statistics over runtime executions of substantial value to the field of runtime verification itself, but also has considerable applications in very interesting and valuable fields.

4.2 Online Statistical Computation

A crucial issue regarding the collection of statistics over runtime executions is the development of an execution strategy for the efficient online computation of the required statistics. In other words, we require a means with which to algorithmically characterise the statistic "behaviour" through the development of the required functions executed at runtime which calculate the statistic valuation. Now, statistics can be considered to be the interpretation and analysis of a set of data. Hence, this implies that in general the computation of statistics requires the manipulation of a substantial amount of data. However, given that statistics collection is augmented upon runtime verification, we require the monitors responsible for statistics collection to be as lightweight and consume the least amount of memory as possible. In order to achieve efficient statistics computation we characterise the notion of incrementally computable statistics defined below (analogous to the definition given in [14, 17]).

Definition 4.2.1 *Given a sequence* $\langle\, s_1.s_2, ..., s_n \,\rangle$*, initial statistical valuation* v_0*, incrementally computable statistic* α *characterised by function f(a,b), where a is a new valuation added to the statistic's input value set, and b is the current statistic valuation, the final valuation v resulting by the application of* α *to* $\langle\, s_1.s_2, ..., s_n \,\rangle$ *is*

$$v \;=\; f(\, s_n, f(s_{n-1},\, ...,\, f(v_1,\, v_0)))$$

In other words, an incrementally computable statistic evaluates the new statistical valuation using only the current statistical valuation and the new value. Note that we shall assume that the forward computation (s_1 to s_n) is equal to the backward computation (s_n to s_1). In fact, a good portion of useful statistics are incrementally computable, such as the average, sum and count. In the case of the sum which returns the addition of the items within the data sequence, the initial statistical valuation is 0, the characterising

function f = $v_i + s_i$, and the data sequence contains the actual valuations which we wish to sum. On the other hand, another set of statistics exist whose nature impede their incrementally computable characterisation. Such statistics include the median, histograms and modes [17].

Given the general statistical framework which we wish to provide capable of expressing a wide variety of statistics, we require the framework to be able to specify both incrementally computable as well as non-incrementally computable statistics. Hence, this issue shall be referred to later on during the statistical framework design. Another issue worth noting is that the issue of incremental and non-incrementally computable statistics directly relates to the issue of system overhead with respect to runtime verification. Whereas incrementally computable statistics simply require the storage (within memory) of the current statistical valuation and occasionally compute the new statistical valuation, non-incrementally computable statistics pose a much higher memory requirement, since all required information for their computation needs to be stored. As discussed in [17], research is being made in solving the issue of efficient non-incrementally computable statistics calculation through the use of randomized algorithms allowing for good fast approximations. However, the exact efficient computation of non-incrementally computable statistics remains an issue.

4.3 Existing Approaches

Although the field of runtime verification has received considerable research in recent years [12, 18, 6, 11], it is perhaps surprising that there is a lack of research into the augmentation of runtime verification with statistics, especially when considering the vast applications (as motivated previously). In fact, the only attempt encountered at developing a theoretical framework for the collection of statistics over runtime executions is [17]. Also, only two currently available runtime verification tools offer support for the expression of statistical queries, these being EAGLE [6] and LOLA [14]. The remainder of the following section involves a discussion of the state-of-the-art through the investigation of these approaches.

Finkbeiner et al. [17] present a framework solely focused on the collection of statistics over runtime executions. This framework is presented through a specified logic extending linear temporal logic [10], and is also considered to be a generalisation of MINMAX CTL [15]. This logic is said to evaluate queries on a trace, whereby queries are constructed from experiments specifying basic observations at individual state locations, and aggregate statistics which combine the evaluation of multiple experiments. Informally, the logic considers the system trace as a list of states, whereby each state contains the valuation to a set of variables.

The notion of the experiment is recursively defined, whereby the core base experiment is the state expression, expressed as a condition, expression pair (p, δ). Given a state expression, a system trace and a position j in the system trace, the expression (within the state expression) is evaluated on the state at position j in the system trace only if the corresponding condition evaluates to true on the same state. If the expression returns false, the experiment is said to fail. Moreover, given experiments ψ_1 and ψ_2, binary function g, unary function f and constant c, the experiment is inductively defined as follows

- $\psi_1 \ \wedge_g \ \psi_2$ — The conjunction operator. Assuming both ψ_1 and ψ_2 don't fail (where both are applied to the same state dictated by the supplied state position j), evaluate both experiments (thus yielding two separate valuations), and evaluate binary function g on both valuations. If either constituent experiment fails, the whole experiment fails.

- $\psi_1 \ \vee_g \ \psi_2$ — The disjunction operator. Assuming either ψ_1 or ψ_2 doesn't fail, evaluate both experiments and evaluate g on both valuations. If both constituent experiments fail, the whole experiment fails.

- $\neg_c \ \psi_1$ — The negation operator. If ψ_1 fails, return c. Else, return fail.

- $\bigcirc_f \ \psi_1$ — The next operator. Given that the operator is applied at position j in the system trace, if ψ_1 doesn't fail on the state at location j + 1, evaluate ψ_1 on the state at position j, and apply unary function f on the result.

- $\psi_1 \ \mathcal{U}_f \ \psi_2$ — The until operator. Given that position j (in the system trace) is the earliest position where ψ_2 succeeds, provided that psi_1 continually succeeds until that point, evaluate psi_2 on the state at position j, and apply f to the result. If no such j exists, return failure.

Note that although some experiments require information from succeeding states, all experiments ultimately return the valuation when applied to particular states. Hence, given this tool for the extraction of observations at individual trace locations, we require a method with which to combine all observations into one statistical result. This is achieved through the aggregate statistic, and is once again recursively defined. The base case is the experiment, hence implying that statistics can be defined on singleton state locations. Given aggregate expressions γ_1 and γ_2, binary function g, function α and condition φ representing the implementation defining an incrementally computable statistic, the aggregate statistic is defined as follows

- $\gamma_1 \ \wedge_g \ \gamma_2$ and $\gamma_1 \ \vee_g \ \gamma_2$ — Conjunction and disjunction operators whose semantics is analogous to that defined for experiments.

- $\mathcal{C}_\alpha\ \gamma_1$ — The unconditional collection. Given that the statistic is applied to state at location j, evaluate the aggregate statistic till the end of the trace using function *alpha*.
- $\gamma_1\ \mathcal{I}_\alpha\ \varphi$ — The interval collection. Given that the statistic is applied to state at location j, evaluate the aggregate statistic using function *alpha* for the duration that *varphi* evaluates to true.

It seems clear that the above definitions for the evaluation of aggregate statistics require that the statistic is incrementally computed, since each experiment is progressively evaluated once, with the new experiment valuation integrated with the statistic valuation resulting up till the currently traversed state. In truth such an evaluation strategy is essential if statistical evaluation is to be efficient. The following are a few statistic declaration examples using the above specified logic (assuming each state in trace σ corresponds to a valuation of variables x & y).

$$[\,\mathcal{C}_{\text{count}}\ (x = 2y)\,]_{(\sigma,0)}$$

which returns the count of states whose valuation of x is twice that of y.

$$[\,(\,\text{true}\ :\ y)\,\mathcal{I}_{\text{min}}\ (x\ =\ 0)\,]_{(\sigma,0)}$$

which returns the minimum value of y for the interval where x is 0 (starting from location 0 in the system trace). Given the above logic we now require a mechanism for its efficient evaluation. Hence, Finkbeiner et al. also present algebraic alternating automata, an extension to alternating automata [18], whereby the presented logic is converted to a semantically equivalent algebraic alternating automaton linear in size to the specification. An algebraic alternating automaton defines no notion of acceptance or rejection, only of evaluation. Its definition admits two distinct node types; terminal nodes labeled with a condition and an expression, and transient nodes labeled with a unary function and a next-state relation. Nodes can also be conjoined or disjoined, whereby each conjunction and disjunction is associated with a binary function. For a more complete definition of algebraic alternating automata see [17]. Of particular interest to us is the evaluation strategy required for the efficient evaluation of algebraic alternating automata. In fact, forward evaluation (starting from the first state in the trace and moving forwards) is exponential in the length of the trace due to the branching nature of the logic. However, this intractability can be avoided by a backwards evaluation strategy (starting from the last state in the trace and moving backward) [17]. A crucial implication of this restriction on the evaluation strategy is the fact that the whole event trace must be available beforehand, thus implying that the presented approach can only be applied to offline monitoring (since by definition, during online monitoring the trace is being generated on the fly, and the last state in the trace is unknown until the system terminates).

4.3.1 LOLA

LOLA [14] is a functional stream computation language, similar in nature to synchronous languages such as LUSTRE [19] and allows for the specification of properties both in the past and in the future. Although the defined logic is succinct and parsimonious in its use of constructs, it is in fact very expressive, exceeding traditional logics used for with runtime verification such as linear temporal logic and finite state automata. LOLA is based on a logic whose constructs enable the expression of correctness-related specifications, as well as statistical properties and related numerical queries. Monitors implementing LOLA specifications can be executed both offline as well as online. Moreover, the LOLA framework guarantees bounds on memory requirements for a syntactically characterised subset of the available logic, as shall be discussed below. This is a crucial issue with regards to the implementation of efficiently executable monitors.

A LOLA specification essentially describes a mapping between output and input streams through the specification of the required computation defining this mapping. Hence, the execution of a LOLA specification computes arithmetic and logical operations over a set of input streams producing reports regarding property violations, as well as statistical valuations. The constructs defined by a LOLA script are a set of typed input streams, a set of typed stream expressions and a set of triggers. Input streams represent input values extracted from the system, and hence require no further definition. However note that each input stream has an associated type (such as integer or real). A valid stream expression is said to define an output stream, and is constructed as follows:

- As a constant of type T, whereby such a stream expression is said to be atomic.
- As a stream variable of type T whose specification is equal to a stream variable (another input or output stream). Such a stream expression is also atomic/
- If f is a k-ary operator of type T, then f(e_1,...,e_k) (where e_1 to e_k are stream expressions) is a stream expression of type T.
- If b is a boolean expression on stream variables, and e_1, e_2 are stream expressions both of type T, then ite(b, e_1, e_2) is a stream expression of type T. The semantics of ite is identical to the if-then-else conditional, whereby if b evaluates to true, e_1 is evaluated, whereas if b is evaluated to false e_2 is evaluated.
- Given stream expression e of type T, constant c of type T, and an integer i, then e[i, c] is a stream expression of type T. In simple terms, this construct refers to the valuation of stream expression e offset by i positions from the current position in the past as well as the future. If i takes the the expression beyond the trace limit, it is reset to the value c. Hence, this construct allows for the expression of real and past time temporal logic.

On the other hand, given a boolean condition φ specifying a condition on stream expressions, a trigger is defined as

$$\text{trigger } \varphi$$

where upon the constituent stream expressions triggering φ, the trigger is raised. Triggers take the form of warnings or errors within the execution of monitors representing the LOLA specification. The following is an example LOLA specification stating that the occurrence of event a must always be at least two more as that of event b (analogous to the example given in [14]).

$$s = s[-1,\, 0] \;+\; ite((a \wedge \neg b),\, 1,\, 0) + ite((b \wedge \neg a),\, -1,\, 0)$$

$$\text{trigger}(s \leq 1)$$

The occurrence of event a increments stream s by 1 while the occurrence of stream b decrements it by one. Hence, given the previous stream evaluation and the event which updated the stream evaluation, if the count does not exceed 1 this implies that the distance between the number of occurrences of events a and b is not at least two, hence firing the trigger.

The algorithm for online monitoring executing queries expressed using the above logic follows a partial evaluation strategy. Concisely, the algorithm executes by incrementally constructing output streams given the available evaluated expressions and input streams, while also keeping maintaining a list of partially evaluated expressions for expressions which make reference to past/future valuations of the stream. In truth, this unbounded list of partially evaluated stream expressions implies that a set of stream expressions defined using the previous logic are inefficient, due to the requirement of an exponential amount of memory (depending on the size of the trace) for the storage of unresolved equations. Hence, the LOLA framework syntactically characterises a subset of the logic for efficiently monitorable specifications, whose resulting memory requirement is linear in the size of the specification and constant in the size of the trace.

4.3.2 EAGLE

EAGLE [6] is a language independent tool and logic used for runtime verification, and requires a user defined projection of the underlying system state (implying manual instrumentation). This state is to declare all variables of interest, whereby formulae defined in an EAGLE script are evaluated with respect to this projected state. The EAGLE logic essentially extends μ-calculus with data paramtrisation, and offers a concise yet expressive set of constructs essentially allowing for the expression of recursive parameterised expres-

sions with minimal and maximal fixed point semantics. Although the specified language is concise (defining three constructs), it is in fact sufficiently expressive to encode a wide variety of formalisms. Such formalisms include future and past linear temporal logic, extended regular expressions, finite state automata and even statistical languages (through the use of data parametrisation). Hence, EAGLE also allows for the use of multiple such logics in parallel.

EAGLE is a rule based, finite trace approach defining three constructs; the next-time ($\bigcirc$), the previous-time ($\odot$) and the concatenation constructors. An EAGLE script defines a set of rules and monitors. Monitors are the actual expressions which are monitored for correctness, whereas rules are used to encode other logics using the defined EAGLE constructs as well as other rules (hence allowing for recursive definitions). Each operator (defined by a rule) can be defined as having a maximal or minimal fixed point interpretation. Hence, if by the end of the trace a formula defining an operator marked with a maximal fix-point interpretation has not yet been violated, the formula will evaluate to true. On the other hand, a formula marked with minimal fix-point semantic will evaluate to false if not yet validated by the end of the trace. The following are example EAGLE definitions (based on examples given in [6])

max Always(Term F) = F $\wedge$ $\bigcirc$Always(F)
min Eventually(Term F) = F $\vee$ $\bigcirc$Eventually(F)
mon M_1 = Always((x = 0) $\rightarrow$ Eventually(y = 0))
min R(int num) = Eventually(y = k)
mon M_2 = Always((x > 0) $\rightarrow$ R(x))

The first two formulae express temporal operators always ($\Box$ F) and eventually ($\Diamond$ F). Also note how formulae are parameterised within rule definitions. Monitor M_1 checks that each time variable x is set to 0, variable y is eventually set to zero too. As exemplified in rule defining R, rules can also parameterise data. Using this rule, M_2 is defined to verify that when x is set to a value larger than zero, eventually y is set to this same value. It is through this data parametrisation that rules defining statistical operators can be expressed, as exemplified below. Assume that variable *eventExec* contains a string representing the event name whose event was last executed within the system. Also assume that function "term()" is a function call denoting system termination. The following rule Count counts the number of event occurrences within the system.

```
max Count(String evName, String lastEv, int num) =
    ((lastEv == "term()"") ∧ (num == 0))
    ∨
    ((lastEv != "term()"") ∧
        ((lastEv == evName) ∧ ◯ Count(evName,lastEv, num - 1) ∨
        (lastEv != evName) ∧ ◯ Count(evName,lastEv, num) )
```

where Count returns true if upon termination of the system event passed under the data parameter *evName* occurred exactly *num* times. Parameter *lastEv* passes the last event to have occurred within the system. If the last event to have executed within the system is the terminate function, then the rule returns true if the variable keeping count of the occurrences of the event has been reduced to zero. If the last executed event is not the terminate function we have two options, either this event is the event of interest or not. In both cases we recursively call Count on the next time instant, however if the executing event is the event of interest the count is reduced by 1, else leave the count unaltered.

Note that a programming oriented extension to EAGLE also exists called Hawk [13], and presents a runtime verification tool for Java programs. Hawk is an event based approach (as opposed to EAGLE which is state based), whereby Hawk specifications are directly converted to EAGLE monitors. Although Hawk is restricted to the verification of Java programs (unlike EAGLE which is language independent), Hawk has the advantage of being automatically instrumentable.

4.4 Discussion

The following is a discussion of the analysed approaches toward the augmentation of runtime verification with statistics, keeping in mind the aims and objectives identified previously.

Although the work presented by Finkbeiner et al. in [17] is valid and insightful, we identify certain deficiencies which we shall be addressing. The first issue with the logic is the lack of real time. Hence, useful statistics such as "the number of user downloads within the last hour" or "the costliest item purchased within the last 24 hours" can not be expressed using the presented logic. Another issue is that the defined algorithms for the efficient evaluation of algebraic alternating automata (which the logic is converted to) requires knowledge of the complete trace a priori, implying that query evaluation is only available for offline monitoring. This is a severe restriction, since we require the evaluation of statistics to be done during online monitoring, especially when considering that most applications mentioned above imply the need for online statistics computation. Yet another issue is the fact that the developed framework solely focuses on statistics collection. However, we envisage the need of runtime verification and statistics collection to be interdependent and compute concurrently. Hence, more work is certainly required for the investigation of how both concepts are integrated in such a manner that is fruitful and intuitive. Another issue worth mentioning is the fact that the presented logic offers support exclusively for incrementally computable statistics, which implies that a substantial subset of statistics whose nature is non-incrementally computable cannot be encoded within the logic. Having said that, an idea which seems particularly appealing

is the evaluation of statistics at discrete points, as well as the collection of statistics over intervals, although we envisage the requirement of more expressive interval semantics than the definition of intervals solely through a boolean condition.

LOLA introduces an expressive language capable of expressing a wide variety of statistics for online computation. In particular, the logic is executable both online as well as offline, and its constructs are able to encode a variety of both future and past logics. Moreover, the framework proposes a syntactically characterised subset of the logic for the execution of efficient monitors. Also, an elegant idea proposed which we believe to have potential is that of the trigger, which is able to mark instances when correctness-related properties break, as well as when conditions based on statistical valuations is met. However, the defined logic also implies a certain set of deficiencies, most notably the lack of real time expressivity, as well as the lack of reparation since the monitor only goes as far as triggering an error report. In truth, we also envisage a tighter binding between runtime verification and statistical frameworks than that presented in LOLA. Whereas streams responsible for simulating correctness-related properties and those responsible for statistic evaluation are able to access each other's valuation, hence implying the sharing of information between frameworks, we also theorise the need of both frameworks being able to control each other's execution (motivated below). In fact, as exemplified by the case studies presented in [14], LOLA is mostly applicable to hardware settings such as the monitoring of the memory controller or PCI.

In truth, EAGLE is analogous to LOLA in nature, with the only distinction being LOLA's descriptive nature [14]. Certainly EAGLE's approach is intriguing, especially with the high level of expressivity implied by the logic capable of encoding multiple formalisms and used in parallel. However, we also identify a set of issues with EAGLE, especially when considering EAGLE for the expression of a statistical framework. Firstly, the issue of manual instrumentation may lead to considerable errors and should be avoided, especially given robust instrumentation tools available (such as aspect oriented techniques [26]). However, we do recognise that EAGLE chooses the path of manual instrumentation for language independence. Another issue is the fact that EAGLE does not natively support real time, although it can be encoded within the language (by triggering an event every time unit). Yet another issue is the relative complexity in specifying statistical properties, even those of trivial nature. This issue is especially considerable given the requirement of an intuitively specifiable statistical framework.

Given that no current approach satisfies all that we deem is required of a statistical framework, we identify the requirement for the development of a new approach. Perhaps the two most glaring deficiencies present in all approaches is the lack of integration of statistics collection with real time (which we identify to have considerable practical applications), as well as the lack of integration of the statistical and runtime verifica-

tion frameworks, which although attempted in LOLA, is not deemed sufficient. We also theorise the need of a more intuitive manner with which to specify statistics, which we consider to be non-trivial for most approaches. An interesting observation worth pointing out is the fact that all current approaches to statistics collection has been logic based. However, since LARVA (an automaton based runtime verification framework) has been chosen as the underlying framework which we shall augment with statistical capabilities, a more automaton based approach is more appropriate.

4.5 Conclusions

The augmentation of runtime verification with statistical capabilities is indeed fruitful. It allows for the straightforward specification of non-functional requirements, which have been surprisingly rather overlooked in current approaches to formalisms for the expression of specifications within runtime verification. Also, statistics aid in capturing indicators of a violation rather than wait for the violation itself to happen, which in certain crucial specifications is vital (especially in specifications which once broken may lead to loss of money or resources). Moreover, as previously discussed the application of the field is comprehensive, with other fields such as intrusion detection, user modeling, performance profiling and quality of service all benefitting from a framework which collects statistics in an online fashion. Finally, the augmentation of statistics leads to the increased conciseness of certain specifications, as well as possibly an increased expressiveness of certain logics.

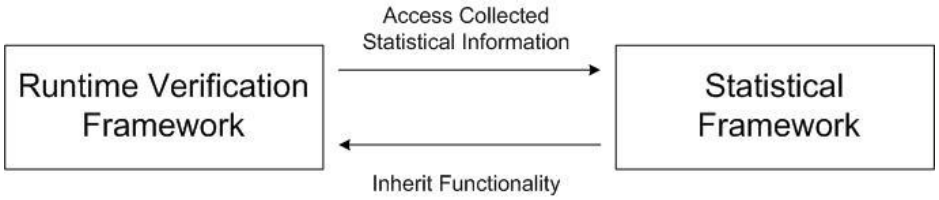
However, although it is clear that such an endeavor is fruitful and worthwhile, there is a need of more research in the area. Three approaches have been identified and have been discussed in detail. However, deficiencies have been identified in all current approaches, the most glaring being the lack of integration of real time with statistics collection, as well as the weak (sometimes completely lacking) communication between runtime verification and statistics which we have identified as vital. Nevertheless, certain valid issues and interesting ideas have also been identified, which serve as an inspiration to the design of our statistical framework.

Part II

LarvaStat

5. The Statistical Framework

The following chapter discusses and motivates the design of a sufficiently adequate statistical framework. The core aims previously discussed in section 1.2 have been kept in mind during the design process. Hence, we recognise the need of a concise yet expressive language capable of expressing a wide variety of statistics. Moreover the focus of this language, apart from the core requirement of expressivity, is that of ease of use and intuitive statistics expression, event at the cost of some syntactic sugar. Finally, apart from the requirement of statistics collection at runtime, the framework should allow for a straightforward and significant means for the connection of the statistical and runtime verification frameworks in such a manner which is fruitful to both. Although this connection will be motivated further below, we envisage a mutually dependent bond between both frameworks, whereby both have access to each other's information and functionality as exemplified below



The diagram above clearly depicts what we perceive as two distinct yet inter-dependant frameworks, with the final result being a framework whose expressivity, applicability and overall suitability is greater than the sum of its parts.

The following chapter motivates the design choices relating to the constructs and overall functionality afforded to the statistical framework through the discussion of a set of identified crucial issues. Hence, we shall firstly motivate the core approach taken for the expression of statistics collection (section 5.1), then proceed to motivate the core constructs defined within our logic (sections 5.2, 5.3, 5.4). We shall extend the core functionality through the definition of actions of statistics (section 5.5), which is followed by a discussion defining the notion of the statistical ordering (section 5.6). Our approach

toward the issues of evaluating non-incrementally computable statistics, as well as the augmentation of runtime verification with statistics is consequently presented in sections 5.7 and 5.8 respectively, which results in the design of the statistical event (section 5.9). Following this, we present an interesting approach toward the definition of multilayered statistics in section 5.10. We then proceed to discuss the impact of the crucial issue of context on the system (section 5.11), and finally conclude by motivating our approach (defined in the following chapter) regarding the operational semantics given to our defined logic (section 5.12).

5.1 Chosen Logic

Choosing the appropriate logic with which to express our constructs responsible for collecting statistical information over runtime executions is no trivial task, more so due to the vast array of logics available capable of influencing the statistical language design. This issue is compounded by the fact that no best logic exists, whereby the choice of logic often depends on the situation at hand. Nevertheless, practical applications to the field, issues with the online computation of statistics, as well as related work in the area serve as a good starting point toward the logic design.

Analogously to runtime verification, we broadly dissect applicable logics into two; logic-based and automaton-based. In fact, whereas previous attempts towards the collection of statistics over runtime executions were entirely logic based [14, 6], the area is yet to see an automaton-based approach towards statistics collection. While both approaches used sufficiently expressive languages for the expression of statistics collection at runtime, both resulted in somewhat complex statistical definitions due to their parsimonious use of constructs. Moreover, certain approaches also lacked certain interesting features such as the expression of real time statistical properties, as well as computation of non-incrementally computable statistics. This leads us to the design of a new statistical language capable of expressing a wider range of statistics, while making as little compromise as possible on efficiency.

Two designs were considered during the design of the statistical framework, one being logic based while the other being automaton based. Keeping in mind the concept of a system trace as a sequence of states which the system goes through, where each state contains the current valuation for each variable (explicit as well as implicit) defined within the system, the following are the considered designs

- **Value Extraction and Value Aggregation Logics** – In a technique somewhat similar to that used in [17], this logic based technique entails the development of two logics. Whereas the value extraction logic is responsible for recognising states

of interest to the statistic, possibly basing an evaluation on that state, the value aggregation logic dictates an evaluation mechanism computing the result given multiple valuations extracted through the value extraction logic. Such a strategy could encode in a straightforward manner issues such as incrementally computable statistics.

- **Point and Update Based Statistic Evaluation** – This automaton based technique is comprised of the following notions (i) The current statistic valuation representing the current statistic valuation at that point in the process of statistic collection, (ii) A point of interest (of interest to that statistic) and (iii) A statistical update function which, given the triggering of a point of interest, evaluates the new statistic valuation. While not immediately obvious, the concept of current valuation and update triggering are analogous to those of state and transition, which is why this logic is considered to be automaton based.

While neither approach were initially formally analysed, certain issues immediately stand out. The main issues of the first approach are those of flexibility and system overhead. While the first approach offers a sufficiently expressive language for the definition of simple statistics, serious doubts remain regarding the definition of more complex statistics, especially with regards to multilayered statistics and non-incrementally computable statistics (as is the case of the analogous work done in [17], where lack of expressivity of such statistics is a well documented issue). Also, the computation of such logics could become unreasonable, due to the theoretically unbounded amount of values generated by the value extraction logic. Turning to the second approach, although the intuitive nature inherent in any automaton based language is a desirable trait, it remains to be seen if such a deceptively simple approach is sufficiently expressive. On the other hand we could take an approach similar to LARVA, whose semantics started from a basic intuitive core, and expanded with additional features in order to make it sufficiently expressive for our needs.

Consequently, we shall be taking an automaton based approach toward the design of our statistical language. This design choice is compounded by the use of DATEs as the mathematical basis of the runtime verification framework, which is an automaton based based framework for the expression of runtime verification properties. Consequently, this approach should aid in the straightforward connection between both frameworks as required above. This remainder of this section is focused on the motivation of the logic used for the expression of our statistical framework, starting from a basic core concept, and proceeding to add the required functionality in an evolutionary manner.

A crucial issue is whether the resulting statistical monitors execute at runtime or offline. Analogously to the choice of logic for runtime verification, such an issue directly affects the choice of the statistical logic, since certain logics require the whole event trace a priori in order to efficiently evaluate statistics (as is evident in [17]). However,

we recognise the vast applications of statistical properties at runtime (as mentioned in chapter 4), especially due to the possible corrective action taken by runtime verification monitors based on online statistical information. In truth, the choice of LARVA limits our options on this issue, since LARVA currently generates exclusively online monitors. Note that henceforth we shall be referring to the notion of an event as defined in LARVA [12], and will hence be represented as an event expression. On a final note, there exists an ever present tradeoff between logic expressivity and the construction of efficient monitors, which we shall also keep in mind.

5.2 The Point Statistic

The point statistic is the primary basic construct defined within our logic encapsulating the core notions of the point and update based statistic evaluation technique (previously motivated). Imagine the situation where we require the specification of a statistic counting the occurrences of a particular event. Certainly, a primary component for the description of such a statistic is the characterisation of the points within the system execution marking when these events are fired. Moreover, we also require a storage location which stores the current count. Consequently, all that is left for the complete executable specification of such a statistic is an update function, which given the current statistic valuation and the occurrence of the event under investigation, increments the count by one. Consequently, the point statistic requires a triple of information

- The current statistic valuation – Each point statistic construct is to have a current statistical valuation of the appropriate type, typically numerical such as integer or real. The current statistical valuation stores the statistic state.
- A point of interest – A point of interest which represents the general instance within the event trace of interest to the statistic.
- A statistic update – An update function which, given the current statistical valuation of all statistics defined within the framework, as well as the current state, evaluates the new statistic valuation. The statistic update executes upon the triggering of the point of interest.

The point of interest represents a core notion within our logic, and is described below.

5.2.1 Point Of Interest

We identify three possible approaches towards the definition of a point of interest within our statistical framework

- **State Based Point Of Interest** – triggering upon the manipulation and alteration of certain values within the system state (Similar technique used in [6]).

- **Boolean Based Point Of Interest** – triggering upon the satisfaction of a boolean expression on the current system state (Similar technique used in [17]).
- **Event Based Point Of Interest** – triggering upon the occurrence of an event within the underlying system (Similar technique used in LARVA[12]).

Although the boolean based point of interest can be considered as an augmentation of the state based equivalent, both run the risk of not being efficiently monitorable since monitoring variables poses a high computational overhead on the underlying system. Therefore, we shall be using event based point of interests within our framework, similarly to LARVA which should aid in the connection between the two frameworks. Consequently, a point of interest shall be triggered upon the triggering of an event expression within the underlying system. Moreover, we need a mechanism which enables us to differentiate between events, since certain events depend on certain context in order to deduce the nature of that event. Consequently, we shall augment the event based point of interest with a boolean condition, resulting in the point of interest within our framework being defined as a pair of information

- An event expression
- A boolean condition which is to evaluate to true if the point of interest is to trigger.

Note that boolean condition is to have access to (i) the set of current statistical valuations within the statistical framework (ii) the event parameters which triggered the point of interest, as well as possibly the return value (iii) the current system state (iv) constructs defined within the runtime verification framework (such as DATEs, clocks and variables).

5.2.2 Statistic Update

The statistic update essentially encapsulates the algorithm responsible for updating the statistic evaluation. Consequently, given the triggering of the statistic's point of interest (implying that the statistic requires updating), the statistic update is executed in order to keep the statistic valuation up to date. Of particular interest is the observation that this model of statistic evaluation allows for a straightforward encoding of incrementally computable statistics as described in section 4.2. On the other hand, this model does not allow for the encoding of non-incrementally computable statistics, which is an issue which will be dealt with later on.

Although statistic evaluation often degenerates to counting or other trivial scenarios, it is also sometimes the case that statistic updates require context information from the underlying system. Therefore, the statistic update function is to have access to (i) the set

of current statistical valuations within the statistical framework (ii) the event parameters which triggered the point of interest, as well as possibly the return value (iii) the current system state (iv) constructs defined within the runtime verification framework (such as DATEs, clocks and variables).

System events are monitored within the statistical framework. Upon the triggering of the event expression defined within the point of interest (i) if the corresponding boolean condition is to evaluate to true; (ii) the statistic update function is evaluated, thus updating the statistical valuation.

5.3 The Interval Statistic

The interval statistic is the second construct defined within our logic which augments the notion of the point statistic through the addition of the interval. Whereas the point statistic assumes that the whole event trace is of interest to the statistic, the interval statistic allows for the definition of a point statistic over a subsequence of the event trace. This is achieved through the definition of an interval characterising the event trace subsequence of interest to the statistic. As an example, defining a statistic counting the total amount of bytes received for each download connection implies that the only event trace subsequence of interest to this statistic is that subsequence during an open download connection. In general, the following diagram summarises the differences between the point and interval statistic.

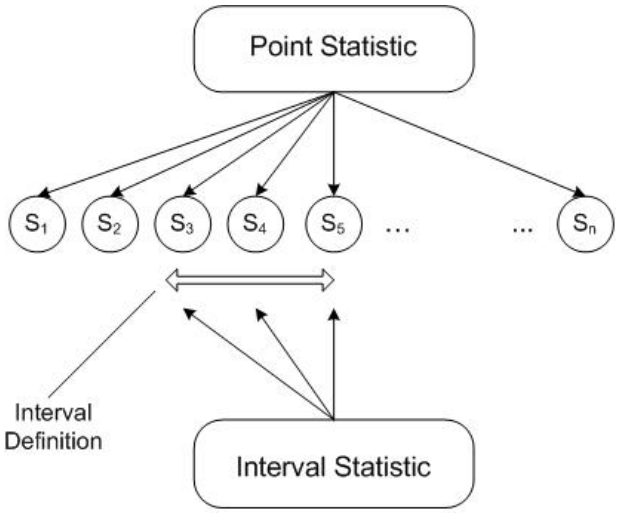


Figure 5.1: Comparison of the point and interval statistic

Since the statistic evaluation strategy within an interval is identical to that for a point statistic, the remaining issue is the choice of interval semantics which will be discussed

in the following section. The choice of a sufficiently expressive interval logic is crucial, since we consider the interval statistic as the backbone of our statistical framework due to the augmented semantics with respect to the point statistic. Moreover, the definition of intervals over the system execution greatly enhances the overall logic expressivity.

Another take on the relationship between the point and interval statistic is the similarity with the approach taken in [17], more specifically the relationship between the unconditional and the interval collection. However, as will be discussed in the next section, the approach taken for our framework is more expressive, since whereas the interval collection[17] defines its interval semantics through a boolean expression, we shall be defining far more expressive interval semantics.

In short, an interval statistic can be considered as a point statistic over a limited event trace subsequence, or conversely that a point statistic is an unbounded interval statistic. This leads us to the following definition of the interval statistic

- The current statistic valuation – The current statistical valuation of the appropriate sort.
- A point of interest – A point of interest which represents the general instance within an interval of interest to the statistic.
- An interval definition – The interval defining the subsequence of the event trace of interest.
- A statistic update – An update function which, given all the current statistical valuations within the statistical framework as well as the current state, returns the new statistic valuation upon the triggering of a point of interest.

whereby the semantics of the statistic valuation, point of interest and statistic update are identical to point statistic counterparts. An additional note worth pointing out is the relationship between the point of interest and interval definition, whereby the point of interest can only trigger if the interval definition deems that the current state (within the event trace generated at runtime) is considered within an interval. This implies that if the point statistic is to trigger and the statistic update executed, the statistic must be within an interval. Conversely, the triggering of a point of interest while the interval statistic is not deemed to have an open interval results in the point of interest being ignored.

5.4 Interval Logic

The following section motivates the designed interval logic within our framework. Three distinct interval types have been identified which we would like to express within our logic.

- **Boolean expression based interval** – An interval characterised by a boolean expression, whose semantics imply that an interval is open as long as the boolean expression is satisfied (similarly to the semantics of the interval for the interval collection [17]). An example interval statistic requiring such an interval type is a statistic which collects the download count for a user whose user ID is not the root (userID > 0).
- **Interval with fixed points of interest** – An interval whose opening and closing is defined by points of interest, similar to the previously motivated point of interest but defining instances which are of interest to the interval (as opposed to the statistic). Such interval points of interest are used to characterise events of interest to the interval, more specifically the opening and closing of the interval. An interval is hence characterised through the definition of the opening and closing interval points of interest, and lies between these two points of interest. An example statistic requiring such an interval definition is one which collects the number of bytes sent for a download connection, where the interval opening point of interest would be the creation of the connection, while the closing interval point of interest would be the connection termination.
- **Time-based interval** – An interval whose duration lies from a time t prior to the current time, up to the current time. This implies a dynamic interval nature, moving in time without any fixed point of interest. An example statistic requiring such an interval is one which keeps the download count for the last hour.

The designed interval logic is influenced by the example intervals motivated above. Analogously to the statistical logic design, the choice of interval logic hinges not only on expressivity, but also on intuitive interval expression. This implies the consideration of some syntactic sugar in order to aid in interval definition clarity and conciseness.

A basic approach towards the expression of intervals is through propositional logic, as done by Finkbeiner et al. in [17]. We postulate that however useful, expressing an interval using such basic means is not sufficiently expressive. Hence, we shall be using propositional logic in conjunction with other interval types.

The first approach taken within our interval logic is through the characterisation of intervals using fixed points of interest, and is sufficiently expressive to defined a wide variety of definitions.

5.4.1 Interval Point Of Interest

A very convenient and straightforward manner with which to characterise an interval is through the characterisation of conditions required for the opening and closing of the interval, and shall be henceforth referred to as interval points of interest within our logic.

As the name implies, such points of interest are analogous to those previously defined for the point and interval statistic. However, whereas the previous definition characterised points of interest to the statistic (signifying that the statistic required updating), the interval point of interest characterises instances within the event trace of interest to the interval. The interval point of interest is defined as follows

- A point of interest
- An action

Upon the triggering of the point of interest, the associated action is executed. This implies that an interval point of interest augments the point of interest with an action. Note that since a boolean condition is a component of the point of interest, this allows for the encoding of boolean expression based intervals within the interval point of interest. The additional action component allows for the execution of some code upon entering or closing the interval, and is usually used for interval initialisation or finalisation purposes. In truth, we recognise the fact that the point of interest semantics requires an associated action which is to be executed upon the triggering of the point of interest. However, whereas the statistic point of interest is associated with the statistic update action, the interval point of interest requires an additional component for the encoding of this action.

5.4.2 Event Interval

The event interval is the first interval type defined within our logic, and is described in the diagram below.

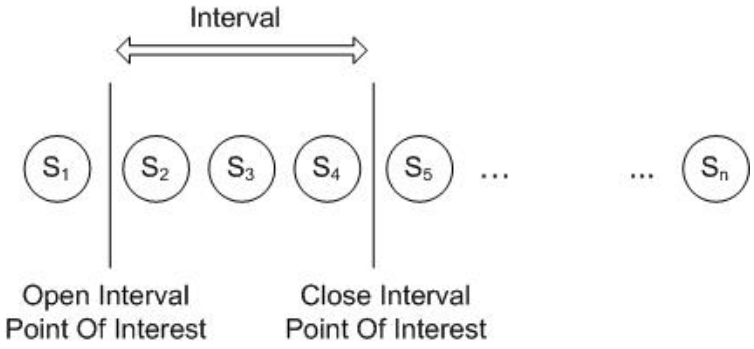


Figure 5.2: The event interval

whereby the event interval is characterised through the definition of an opening and closing interval points of interest. This interval type represents the general form of the previously discussed interval with fixed points of interest, and is sufficiently expressive to encode a large subset of the required intervals used in practice. Moreover, execution of the event interval is a trivial task, since all that is required is to listen for the triggering of the points of interest, and executing the associated actions.

5.4.3 Duration Interval

Although the event interval is a flexible means with which to express intervals, it lacks the notion of time. The second interval type within our logic, the duration interval, represents a variant of the event interval through the addition of real time capabilities. We recognise two distinct possibilities, as specified below.

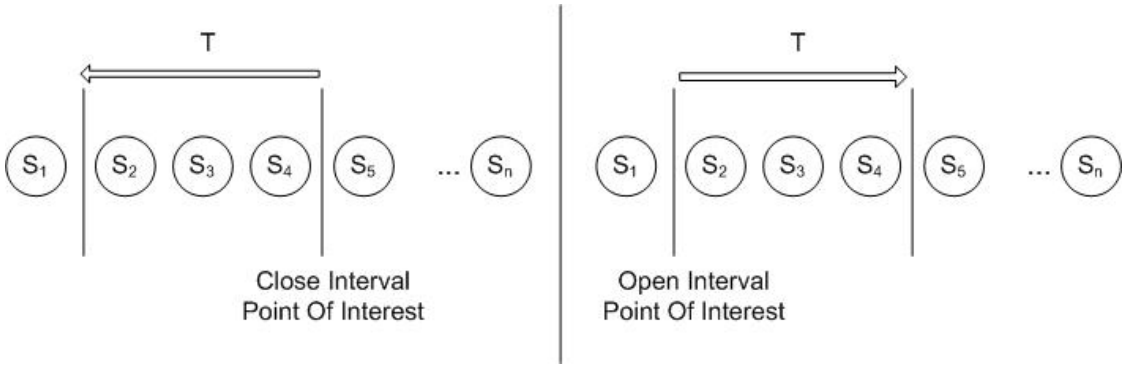


Figure 5.3: Duration intervals

whereby a duration interval is characterised by a fixed interval point of interest and a duration of time t, implying that the interval is open for time T prior to/after the triggering of the fixed interval point of interest. The duration interval offers interesting new expressive capabilities allowing for the straightforward expression of a different class of intervals, used in interval statistics such as for example in the definition of a statistic which collects user information up to a certain time after login (representing the opening interval point of interest).

Although allowing for the expression of both interval duration types expressed above would be a good option to have, the leftmost option must be discarded from our interval logic due to doubts over the efficient online monitoring of past time interval logics. Implementing such a time logic as required above would require a monitor to store the preceding event trace up to a certain time duration, which is unfeasible. In truth, the practical need for such a construct rarely arises, and at worst the requirement of such a construct could be easily restructured so as to avoid the past-time logic aspect of the interval (similarly to what was done in [14]). This resulting duration interval is characterised through the following triple

- An opening interval point of interest.
- A real number defining the interval duration.
- An action triggered upon the closing of the interval.

whereby upon the triggering of the opening interval point of interest, the interval closes upon elapsing the specified duration of time. Upon closing the interval, the associated action is executed.

Since the definition of the event expression as defined in LARVA [12] allows for the declaration of timer events, the duration interval is in fact reducible to the event interval by declaring the closing interval point of interest as a timer event which triggers after elapsing the specified time duration. Nevertheless, although not adding in expressivity the addition of the duration interval is still worthwhile due to the straightforward expression of an additional set of intervals.

Execution of the duration interval is trivial, since we require to listen for the triggering of the opening point of interest, and closing the interval after elapsing the interval duration. Upon closing the interval, the associated action is executed. The duration of time is measured through the use of the timer construct provided by the DATE framework.

5.4.4 Time Interval

The time interval encapsulates the previously defined notion of the time-based interval. As previously hinted, the time-based interval implies a dynamic interval nature which moves in time, which makes it different to the previously defined interval types. The time interval is described below

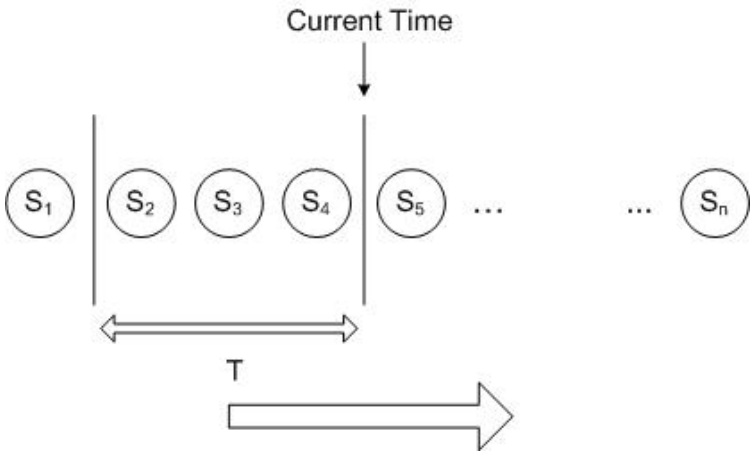


Figure 5.4: Time interval

which implies that the time interval operates with a time frame duration T, and continually moves in order to keep up to date with the current time. Such interval semantics offer the capability to represent a whole class of different intervals, and is very useful when expressing interval statistics such as the number of downloads in the last hour, or

the number of infringements within the last month. Note that the time interval semantics lack the requirement for the definition of fixed points of interest, due to the dynamic interval nature.

Whereas the execution of the other two interval types was a trivial task, executing the dynamic time interval poses certain non-trivial issues, the most crucial being the issue of the silent event. Take the following two properties assuming statistic downloadCountLastHour counting the number of downloads which have occurred within the last hour.

$$\pi_1 : \text{downloadCountLastHour} \leq 10$$
$$\pi_2 : \text{downloadCountLastHour} \geq 10$$

The crucial issue regarding these properties is their behaviour as time passes, given a certain setting. Take the situation where at a particular instant during the course of the program execution property π_1 is satisfied and no further downloads commence. The satisfaction of π_1 implies that the number of downloads within the last hour is less than or equal to 10. Crucially, as time passes (and no further downloads commence) π_1 remains satisfied. This implies that certain statistical properties, upon their satisfaction and no further event expression triggering the associated statistical updates, remain satisfied. Such statistical properties are hence said to be based on a silent event. Concisely, the behaviour of a silent event is said to be invariant to the passage of time. On the other hand, given the satisfaction of property π_2 and the same setting as that given to π_1 (no further downloads), property π_2 is guaranteed to fail with the passage of time. Although downloadCountLastHour is initially larger or equal to zero (since π_2 is initially satisfied), given no further download the statistical valuation decreases in time reaching zero upon elapsing exactly one hour from the last download. This implies that there comes a particular instance where π_2 breaks (when downloadCountLastHour decreases from 10 to 9), without any system event being directly responsible for the property breaking. In other words, whereas statistical properties based on event or duration interval statistics adhere to a behaviour which is reactive to the system events, statistical properties based on time interval statistics need to be proactive in order monitor property evaluation.

Although the silent event might at first seem tangential, it offers an insight into the core of what makes the time interval semantically distinct from the previous two interval types. Whereas the execution of the statistical update (altering the statistic valuation) for the event and duration intervals can be directly attributed to the triggering of a point of interest within the system, there exists no such mapping for the alteration of the statistical valuation of a time interval statistic. Instead, the time interval requires a dynamic statistical update in order to keep the statistic up to date with the constantly moving dynamic interval. Two candidate mechanisms have been identified for the execution of this dynamic statistical update, as described below.

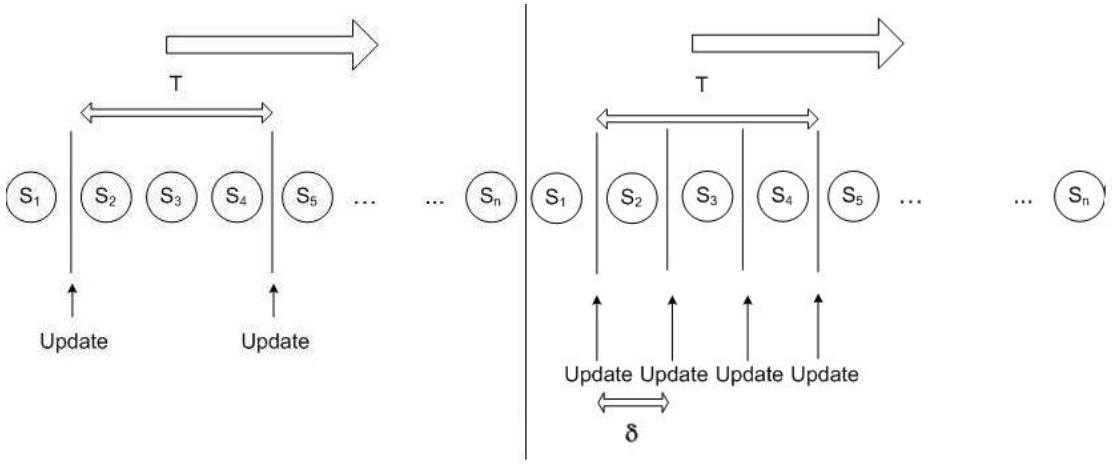
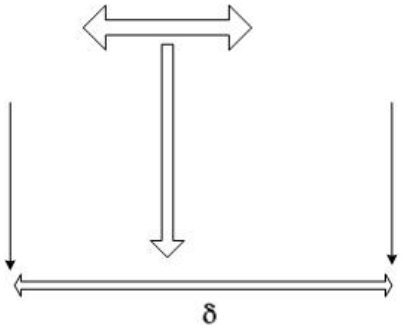


Figure 5.5: Updating mechanism using scheduled updates vs. polling

The leftmost diagram summarises the updating mechanism using scheduled updates, whereby given a time interval of duration T, executes an updating action each T time units. This action is responsible for updating the statistical valuation in order to keep the statistic up to date with the interval semantics. Although such an updating mechanism is efficiently monitorable, it does not perform sufficiently well since using such a mechanism would imply that the statistical valuation is up to date only every T time units. This problem becomes especially unreasonable for time intervals of large durations (such as every hour).

This leads us to the polling mechanism, which is essentially a generalisation of the previous mechanism. Instead of updating every T time units, the interval continually polls the statistic valuation by triggering an action at a much faster rate with a duration δ between each update triggering (as seen in the diagram). The semantics of the action triggered is identical to before, and is hence responsible for (but not restricted to) updating the statistic so as to ensure that the statistic valuation is in line with the time interval semantics. Consequently, the effectiveness of this approach hinges on the size of δ. Theoretically, δ can be infinitesimally small, and the smaller δ is, the more will the statistical update be up to date. However, in practice choosing the value of δ must reach a compromise between computational overhead on the system and the total time kept updated. In other words, since each executed action consumes computational resources, making δ too small will result in the action continually executing and thus incurring a substantial computational overhead on the underlying system. However, making δ too large will result in the statistic not updating fast enough and thus having an outdated valuation most of the time. We shall leave the choice of the size of δ up to the developer of the statistical framework.

The choice of polling as the underlying update mechanism for the execution of time intervals poses an interesting issue. Since the time interval requires the execution of an action every δ time units updating the statistical valuation thus keeping the statistic up to date with the interval semantics, this implies that given any instant, a time interval statistic is out of date at most δ time units, as described below



This also implies that for an interval statistic whose interval is a time interval of duration T, at any one instant the actual statistic valuation at worst evaluates to the time interval of duration T + δ. Although not ideal, we believe this to be the best method to execute such rich interval semantics (as those given to the dynamic time interval) in a manner which is computationally efficient. In truth, a time interval statistic can become rather computationally inefficient unless both the action and the statistic update are properly designed, since during its execution the time interval statistic requires the execution of two actions, one which continually updates the statistic in order to keep it up to date, and another to update its statistical valuation upon the triggering of its point of interest (the statistic update). Note that the use of the polling updating mechanism hides the immediate use of time interval duration T. However, the use of duration T is encoded within the updating function by keeping the statistic valuation in line with the last statistic updates occurring within the last T time units.

Given the choice of the polling updating mechanism, the time interval is defined through

- A real number defining the time interval duration.
- An action triggered each δ time units keeping the statistic valuation up to date with the constantly moving dynamic time interval.

The concept of dynamic intervals which move with respect to time is perhaps a rather novel concept with vast applications. We currently recognise that although extensively used, time intervals using polling can become rather computationally expensive, and in practice the definition of actions which update the statistic valuation in accordance to the interval semantics is usually non-trivial. It is for these reasons that this concept

of dynamic intervals should benefit from further characterisation and formal analysis in order to design even more intuitive, flexible and efficient monitoring of such intervals. Nevertheless, we believe that this first approach is valid and revealing in its own right.

5.4.5 Overlapping Intervals

With the three interval types within our statistical framework characterised, it becomes apparent that certain intervals may face the issue of overlapping intervals described below

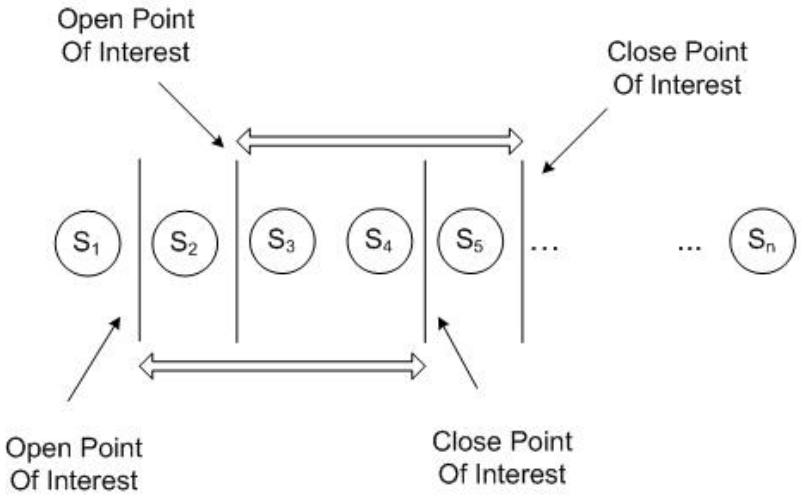


Figure 5.6: Two overlapping intervals

As can be seen two intervals overlap over the same section of the event trace, with two open interval points of interest firing in quick succession followed by two close event points of interest. Such a situation leads to the dynamic generation (at runtime) of two distinct interval statistics, since each interval leads to a different statistic valuation. In practice this situation can frequently arise, especially with multi-threaded systems, and it is for this reason that this situation must be correctly monitored within our statistical framework. As an example imagine the situation where the statistical framework is collecting statistics over open download connections for an ftp server which allows multiple concurrent downloads. Certainly such a scenario allows for users having multiple concurrent downloads, which would give rise to the situation above since the creation of two open connections generates two distinct interval statistics.

While not immediately obvious, the issue of overlapping intervals poses a serious problem to the current constructs defined within our statistical framework. Imagine the situation where two open interval points of interest are triggered, hence creating two interval statistics. The problem lies in the fact that given this scenario, the triggering of a close

interval point of interest will lead to an unresolved situation, since it is impossible to deduce which of the two currently open intervals did the interval point of interest close. The root of the problem lies in the fact that none of the currently defined interval constructs are able to distinguish between interval instances, which leads to the requirement of an associated context.

Note that the above diagram shows only one form of overlapping intervals. In truth the issue of overlapping intervals may take multiple forms, such as totally overlapping intervals (intervals nested within other intervals) as well as intervals whose opening of one interval directly coincides with the closing of the other (similarly to two subintervals resulting from the chop operator defined in duration calculus [8]). Hence, the solution for the expression of overlapping intervals within our framework must be sufficiently flexible for the expression of all forms of overlapping intervals.

Analogously to the introduction of contexts within the DATE framework, we shall be introducing the notion of context to our intervals, whereby if the situation requires (hence if an interval definition may result in the generation of overlapping intervals) each interval definition may be associated with a context in the form of an object, as depicted below.

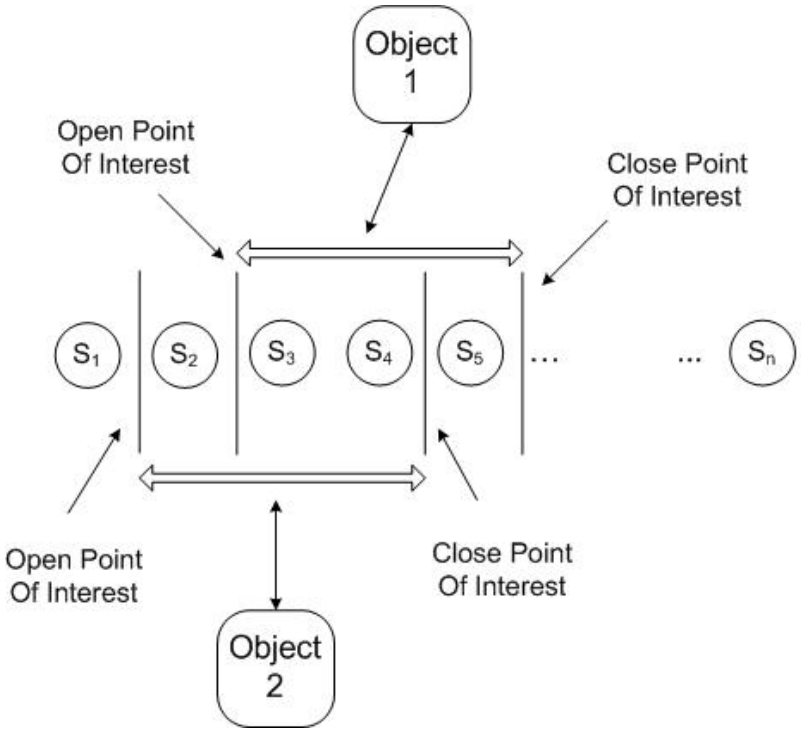


Figure 5.7: Two overlapping intervals

The addition of context allows us to associate an opening interval point of interest with

its closing counterpart, and hence allows us to distinguish between intervals. Continuing from the previous example, had we associated each interval with the connection object created upon the opening of the connection, then the triggering of the closing interval point of interest could easily be resolved by deducing the object upon which the connection termination was executed (and hence which triggered the closing of the interval in the first place).

Although stated without proof, we theorise that this issue of overlapping intervals is only valid for event and duration intervals, and is hence invalid for time intervals. This is because whereas the event and duration intervals require creation at runtime of new interval statistic instances upon the opening of a new interval, the time interval's dynamic nature implies that no new statistics can be created at runtime. In other words, since a time interval moves with time, it is consequently always updated and can never expire. Hence, the lack of runtime interval statistic creation impedes the generation of overlapping intervals.

5.5 Actions On Statistics

Whereas the notion of intervals offer a good tool with which to control the collection of statistics, it can become an overelaborate means with which to achieve simple functionality such as starting or stopping statistics collection. Moreover, we would like to push forward the idea of looking at the statistic within the statistical framework as a superstructure, not too distant in concept from the idea of the object in the object oriented paradigm. This leads us to the choice of operations which shall be permitted upon statistic instances within our framework, which we shall henceforth refer to as actions on statistics. These actions are summarised below

- Reset – Action which resets the current statistical value.
- Change Statistic Value – Action which allows for the manipulation and retrieval of the current statistic valuation.
- Switch Off – Action which freezes the statistic in a suspended state, whereby no further statistics collection is possible. In the case of statistics with associated timers, it is assumed that the timer state is also set to paused. Note that chaining switch off statistic actions has the same effect as one switch off action. Also, executing the switch off statistic action on a statistic which is already frozen has no effect.
- Switch On – Given that the statistic is in a suspended state, this action resumes statistic collection from the exact same statistic state prior to being switched off. If the statistic is associated with any timer, then that timer is also resumed. Chaining

multiple switch on statistic actions also has the same effect as one switch on action. Executing a switch on action on a statistic which is already execution has no effect.

whereby these actions can only be taken into consideration when given the context of a statistic instance. Note that as a design choice all defined statistics are initialised to on. Moreover it is crucial to note that (as will be motivated below) since the runtime verification framework has access to the statistical constructs, all constructs within the statistical and runtime verification frameworks have the capability of executing actions on the statistical constructs. This implies that properties within the runtime verification framework could also theoretically control statistics collection by using appropriate statistic actions.

We theorise that statistic actions do not in fact increase the language expressivity, since switching on or off statistics can be encoded using other constructs such as intervals or boolean conditions. Nevertheless, this feature offers a straightforward means with which to control statistics collection if the situation demands so.

5.6 Statistic Ordering

The general case within our statistical framework can be described as the definition of multiple statistics of varying types and complexity collecting statistics over the currently running system execution in an apparently simultaneous fashion. However, we require a deterministic manner with which to determine the order of the statistics evaluations in order to avoid ambiguity. This need is more apparent in situations where the evaluation of certain statistics depends on the prior evaluation of other statistics. An adequate example is the downloadCountAverage statistic, which uses the statistic evaluation of the totalDownloadCount statistic. Hence, in order to ensure correct evaluation of the average, given that both statistics are triggered on the same point of interest (when a download is successfully completed), the statistical framework must ensure that the totalDownloadCount statistic must be evaluated prior to the downloadCountAverage statistic. This gives rise to the concept of statistic ordering, whereby given that a set of statistics are simultaneously triggered, the following rules for their statistic update execution ordering are adhered to

- Statistics with no dependency relation can be placed interchangeably within the statistic ordering.
- If the statistic evaluation of a particular statistic is dependent on a set of statistic evaluations, then all the associated statistics are to be placed at a position prior to this particular statistic within this ordering.

Clearly this requirement of statistic ordering is only apparent for statistics whose statistic update is triggered on the same point of interest, since the point of interest triggering order 'overrules' any defined statistic ordering. Also note that mutually dependent statistic updates are not allowed within the statistic framework, although in practice none were required.

In order to resolve any possible ambiguity regarding statistics triggering on the same points of interest, we shall extend the notion of the statistic ordering. Thus, apart from the individual statistic definitions, we also require a global statistic ordering adhering to the same rules above such that, given a subset of these statistic declaration which fire simultaneously, the evaluation order is done according to this ordering.

5.7 The Issue of Non-Incrementally Computable Statistics

Whereas the statistic constructs designed within our framework so far encode the computation of incrementally computable statistics in an intuitive and straightforward manner, little care has been taken for the computation of non-incrementally computable statistics. This can be said of other approaches to collecting statistics over runtime executions too, whereby tools such as LOLA [14] specifically state that the logic is not sufficiently expressive to compute non-incrementally computable statistics. We theorise that given our aim of developing a general framework for the collection of statistics at runtime, excluding a substantial amount of statistics whose computational nature is non-incrementally computable, some of which have a variety of applications (such as the median or histograms), would result in a serious limitation to our framework.

An important issue regarding the nature of non-incrementally computable statistics is the requirement of a theoretically unbounded amount of memory, as discussed in section 5.7. This issue poses a great risk toward the generation of efficient monitors, since the monitor computing non-incrementally computable statistics could generate an unreasonable space overhead on the underlying system. However, we shall leave this issue to the designer of the statistic, whereby the designer is to specify the statistic to compute in both a space and time efficient manner. In practice, most non-incrementally computable statistics can be properly designed so as to use a bounded amount of memory, which although larger than that required for the incrementally computable counterpart, is still reasonable.

Similarly to the allowance of additional variable declaration within the DATE framework, we shall allow an unrestricted amount of variable declarations for each statistic definition (both point and interval statistic) which are to be used as temporary storage for additional information required for the evaluation of the non-incrementally computable

statistic. Hence in order to develop efficient statistical monitors, it is up to the statistic developer to design such statistics so as to use a limited amount of variables with a reasonable bound on the required memory.

5.8 Augmenting Runtime Verification with Statistical Framework

As has been previously motivated in chapter 4, augmenting runtime verification with statistical capabilities is very advantageous. However, although the current statistical constructs offer a sufficiently expressive means for the collection of statistics at runtime, additional functionality is required for the actual integration of statistics collection with that of runtime verification. We theorise the need of a tight binding between frameworks, whereby both are able to access each other's information and functionality, and moreover both are also capable of affecting each other's computation, be it properties which alter the execution of statistics collection, or statistics which affect property monitor behaviour. The end result is a new augmented framework, constituting the integration of two distinct yet interdependent frameworks whose overall expressivity and general applicability is greater than the sum of its constituent parts.

Both frameworks have already been carefully chosen, whereby LARVA and the underlying DATEs have been chosen as the runtime verification framework. On the other hand, as previously motivated we have opted for the design of our own statistical framework. We characterise the augmentation of the DATE runtime verification framework with our statistical framework within the following two mechanisms

- The ability for properties to access statistical declarations at runtime, together with access to the associated functionality. This includes access to the current statistic valuations, as well as the ability to execute actions on the statistics.
- Allowing for statistical framework access to the constructs defined within the runtime verification framework, as well as the ability to alter construct behaviour. Such constructs include both monitors within the runtime verification framework and also other statistical constructs.

From the runtime verification framework point of view, we refer to the statistical constructs as additional distinct declarations within the framework, much like clocks or channel declarations. Moreover, similarly to DATEs (representing properties) being able to pause, resume and reset clocks, we shall allow DATEs similar functionality on statistics through the execution of actions on statistics (motivated in section 5.5), thus offering an elegant solution to the first proposed mechanism.

From the statistical framework point of view, each construct defined within the runtime verification framework is accessible to the constructs defined within the statistical framework. This includes clock or channel declarations, as well as declared events. The solution for the enhanced statistical constructs' ability to alter construct behaviour results from the creation of the statistical event, and is described in the following section.

5.9 The Statistical Event

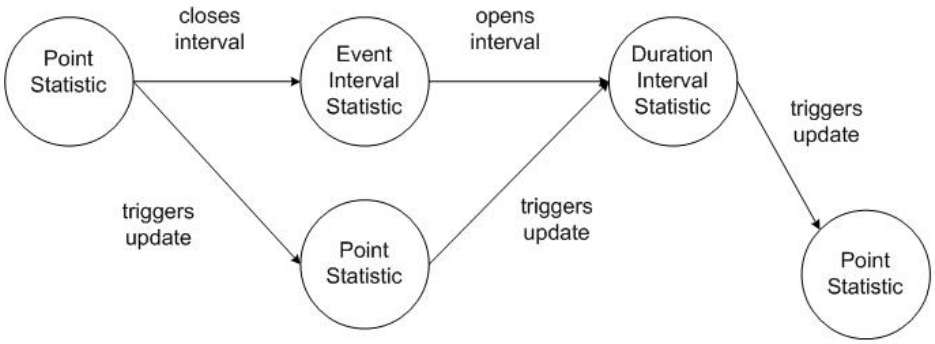
The statistical event is an additional concept within our framework. Whereas the addition of timers within the DATE framework was mainly for expression of real-time properties, and channels were added for automaton synchronisation, we shall view the addition of statistical constructs within the DATE framework as constructs which allow the expression of numerical and statistical properties. Moreover, since both clocks as well as channels augment the concept of the event (within DATEs) by allowing properties to listen to events occurring on these structures, it is a natural extension to augment further the event semantics so as to allow DATEs to listen to events occurring within statistical constructs, which lead us to the creation of the statistical event.

The semantics of the statistical event are straightforward, whereby each time a (point or interval) statistic creates a new statistic valuation an event is generated passing this new statistic valuation within the event. This event is visible to both properties as well as other statistical constructs. Since transitions within DATEs trigger on the firing of events, statistical events allow for the triggering of transitions upon the generation of new statistic valuations. This allows for statistics constructs the capability to alter monitor behaviour as well as the execution of other statistics. Note that unlike other events, the statistical event need not be explicitly defined, since a statistical event is implicitly defined for each defined statistic.

Given the defined semantics, the statistical event can be theoretically simulated using only synchronisation channels, since once a construct is alerted that the statistic has completed evaluation, this construct can retrieve the statistical valuation. However, the importance of the value passing statistical event, apart from clarity, is magnified when statistical valuation access is limited by context rules, as discussed below.

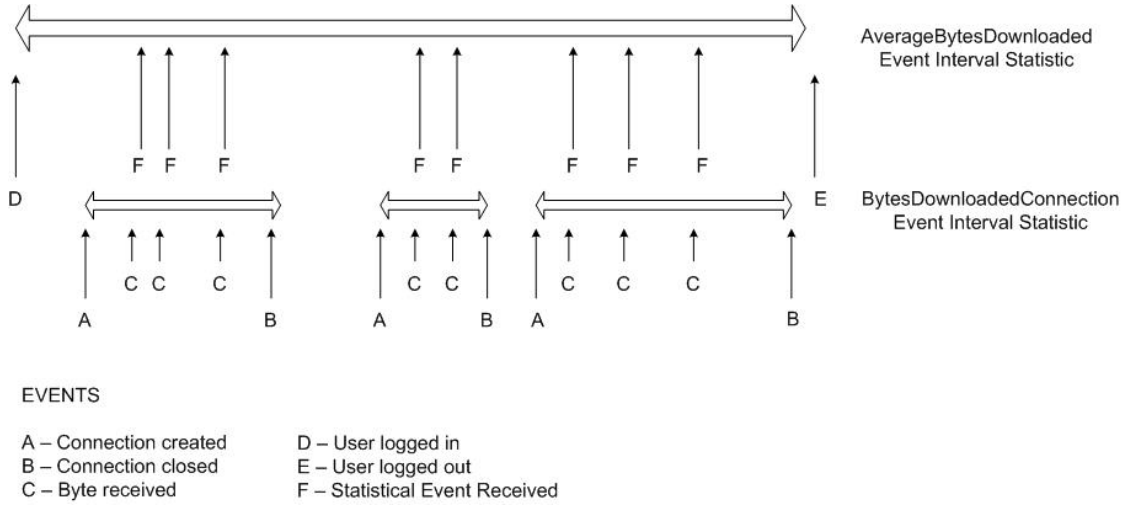
5.10 Multilayered Statistics

Another crucial application of the statistical event is the expression of a multilayered approach to statistics. Since the execution behaviour of statistics is based on the firing of events, statistics can in effect trigger on other statistical events, as exemplified below



Given a statistic which depends on the statistical event of another statistic, we say that the former statistic is dependant on the latter. The flexible semantics afforded to the statistical event allows for a theoretically unrestricted dependency graph representing the dependencies between statistics. However note that cycles within the graph should be avoided, since this could result in an infinite loop of statistical update triggering. Consequently, ensuring that the dependency graph is in fact represented by a directed acyclic graph (DAG) would avoid such potentially hazardous situations. Moreover, continuing on the notion of the dependency graph as a DAG, a topological sort over the DAG returns an ordering which represents the multiple statistical layers resulting from the definition of the statistical dependency described above.

Whereas the point statistic triggers its statistic update on the firing of events, the event and duration interval statistic also fire the opening (in both cases) and closing of the interval (in the event interval case) in event firings, and hence possibly also on statistical events. While not necessarily increasing in expressivity, the addition of the statistical event allows for a more intuitive definition of certain statistics, more specifically nested statistics (statistics within statistics). An example of such a statistic requiring the definition of a multilayered approach is the average amount of bytes downloaded per user session, which can be expressed as two multilayered statistics



The BytesDownloadedConnection statistic is an event interval statistic collecting the byte count downloaded for each connection (assuming the connection is initiated from the same user session), whereby the interval opens upon the connection creation, closes upon the connection termination, and updates its statistic valuation upon the download of a number of bytes. Each time the statistic valuation is updated, a statistical event containing the new statistic valuation is created. Another event interval statistic is defined named AverageBytesDownloaded, whose interval is opened upon user login and closes upon user logout. The statistic triggers its statistic update upon the occurrence of the former statistic's statistical event, and calculates the new average value given the current average and the value sent within the statistical event. Although in this instance only two layers were defined and both statistics were of the same type, one can define an unbounded amount of layers containing different statistic types.

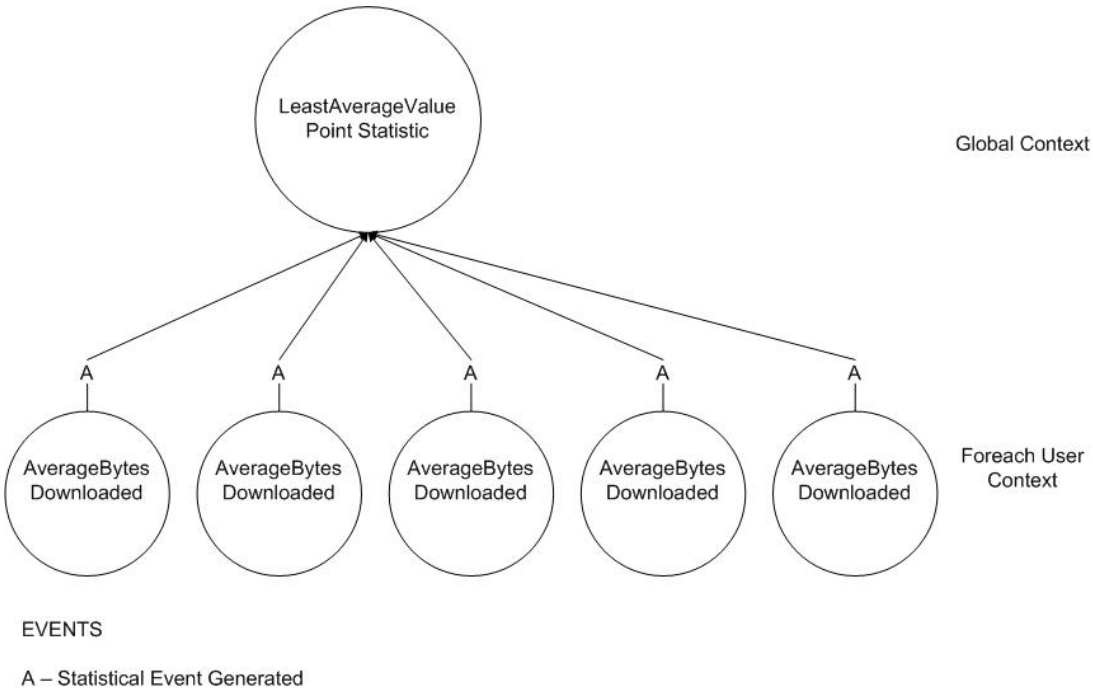
5.11 Context

A crucial issue emerging from the choice of DATEs as the runtime verification framework is the integration of context (as presented in LARVA) with that of statistics collection. As is evidenced in the previous example motivating multilayered statistics, the process of statistics collection itself is not exempt from context. Implicit in the previous example is the fact that each logged in user will have a separate statistic valuation collected during that logged in session. Consequently, we need to be able to distinguish which connection originated from which session so as to be able to update the appropriate statistic. In general, such situations imply the need for the collection of statistics at a per context level (such as for each session).

This approach to context taken by the statistical framework is perfectly in line with that taken by LARVA. Since LARVA allows for the definition of multiple contexts, each

with their local constructs, as a design choice we shall extend this concept by allowing for the definition of statistics at a local level. In other words, each context can also define a set of local statistics obeying the same context rules as all other constructs. Events within LARVA have already been defined with the issue of context in mind which allows for a direct distinction between contexts. This notion of events can also be used by statistics, whereby each event declared within a point of interest is extended to define, apart from the event itself, a context in the form of an object.

We shall also allow the collection of statistics at a global level (disregarding contexts). Frequently context is not required by statistic definitions since the collection of statistics is required at a global level not bound to any sub-context. An example scenario requiring a global context is the collection of the total number of downloads within an ftpd server. In this occasion the collection of statistics does not require a context, since all that is required is to listen to the download complete event within the server, and increment the statistic valuation accordingly. This approach to the collection of statistics at a global as well as at a sub-context level results in a new type of multilayered statistics. Whereas the previous discussion regarding multilayered statistics can still be defined within the same context, we also extend this notion to statistics across multiple sub-contexts. This allows for the intuitive specification of a further set of statistics, as is exemplified in the case of the statistic evaluating the least average number of bytes downloaded



AverageBytesDownloaded represents the two-layered statistic defined in the previous example. An instance of the AverageBytesDownloaded statistic is created for each user

session, and evaluates the average amount of bytes downloaded. The global point statistic (hence not bound to any context) LeastAverageValue triggers its statistic update upon the triggering of a statistical event from any of the subcontexts, and stores the smallest value from all the values sent within the generated statistical events.

One last issue regarding context is the impact of context on the statistical event. Up till now, we have assumed that statistical events trigger within the same context. However, as the previous example shows, we require the extension of statistical events across nested contexts levels. This results in two types of multilevel statistics; within the same context and across nested contexts. The statistical event across contexts allows for communication and statistic valuation passing across nested context levels. Note that whereas explicitly defined events also define the context associated to that event, it remains unclear how the statistical event context is handled for the implicitly defined statistical events. As a design choice, for simplicity we shall be abstracting the explicit declaration of context from the implicit statistical event. Thus, any construct (be it a property or another statistic) wishing to listen to a particular statistical event — regardless of the context it is defined in — can do so without defining the context from which the event is generated, much like timer events on clocks. Clearly each statistic is still associated with an implicit context extracted from the statistic's context. However, context handling is done internally within the statistical framework. Note that circular statistical dependencies (as defined in section 5.10 across contexts should still be avoided.

This design choice of statistical events whose context is implicit across nested contexts implies the restriction of statistics communicating among non-nested contexts. However, we never found any practical use for statistics communication across non-nested contexts. Instead, statistics communication always focuses on statistics across nested contexts. In truth, communication across any context is still achievable through the global context accessible to all contexts. Hence, although statistical events across non-nested contexts is not achievable (although apparently not practically usable), we stand by this design choice due to the resulting simplicity in the use of the statistical event for practical applications.

Statistical valuation access to other constructs is restricted by context rules. However, as previously exemplified it is often the case that statistics at a higher context level wish to access valuations from statistics at lower contexts. It is hence imperative for the statistical event to pass the current valuation within the event, since the statistical event is the only mechanism which allows for the communication of lower level context statistics with their higher level counterparts.

5.12 Execution

With the logic expressing our statistical framework extensively motivated, we shift our attention to the design choice of an execution strategy for this logic. In other words, given the logic, we require a means with which to execute it. The process of defining a means for the execution of a logic is achieved through the specification of an operational semantics. We identify two possible approaches, these being (i) the definition of an automaton based logic which our logic is directly reducible to, or (ii) the translation of our logic defining the statistical framework to DATEs.

Although currently stated without proof, we theorise that DATEs are sufficiently expressive to encode the functionality expressed by the constructs motivated in this chapter. Moreover, the choice of DATEs as our execution strategy allows for the seamless integration of our statistical framework with the DATE framework motivated in section 5.8. This is because the resulting DATEs encoding the statistical constructs can communicate and share information with other DATEs encoding properties as part of the runtime verification framework, and vice versa. Consequently, we shall be choosing the second option of translating our statistical framework into DATE constructs.

As a result of this design choice, chapter 6 motivates the techniques used for the translation of our statistical constructs to DATEs, whereas section 9.2 formalises these techniques.

5.13 Conclusions

Throughout this chapter we have thoroughly motivated an expressive statistical logic which we believe has the edge over related tools used for the collection of statistics at runtime. Of particular note are the linking of statistics with real time aspects, as well as the capability of evaluating non-incrementally computable statistics. Also worth noting is the concept of multilevel statistics which allows for the definition of an unbounded amount of statistical collection layers even across contexts. We believe to have also presented an intuitive yet powerful approach toward the connection of the statistical and runtime verification frameworks. Moreover, the characterisation of such non-trivial concepts were encapsulated within straightforward and intuitive constructs which are translated to efficient monitors.

Given more time we would certainly have liked to study further certain issues of our framework. More specifically, certain areas which we would like to characterise further are the notions of the dynamic interval, as well as the collection of non-incrementally computable statistics. Nevertheless, we believe that our approaches toward these issues are currently sufficiently expressive for their specification within our framework should

they be required. Another important orthogonal issue which has not been given sufficient importance is the overhead impact of our statistical framework on the underlying system. In truth, [12] informally motivated that DATEs are approximately polynomial in size with respect to the size of the specification. Hence, since we shall convert the statistical framework into DATEs, the resulting system overhead approximately depends on the resulting DATEs simulating our statistical framework. Nevertheless, a more thorough analysis of the system overhead implied by each of our statistical constructs is necessary, especially since the risk of inefficient monitors due to non-incrementally computable statistics remains.

Note that statistical declarations across contexts forming the declaration of our statistical framework are to obey two orderings (as previously defined). These orderings include the statistical ordering applicable to all statistical declarations, and the statistical dependency ordering for statistics dependent on other statistics' statistical events. Whereas the statistical ordering offers a guideline for the correct evaluation of multiple statistics declarations, the dependency ordering is present to limit multilayered statistical definitions from possibly entering infinite execution loops.

6. Converting the Statistical Framework to DATE Constructs

As motivated in section 5.12, if we are to execute the defined logic for the expression of our statistical framework we must define an operational semantics for the logic. The approach chosen for this task is through the translation of our logic into DATEs, and implies that we shall execute the semantics expressed within our logic entirely through constructs defined within the DATE framework. These constructs essentially include DATEs, channels and clocks. Implicit in this choice of operational semantics lies the belief that LARVA is sufficiently expressive for the expression of our statistical logic, or conversely that our logic is not more expressive than LARVA.

Three constructs have been defined in all within our language, these being the point statistic, interval statistic and the statistical event. Moreover, the interval statistic defines three different interval types, these being the event, duration and time intervals, all of which have a considerable effect on the interval statistic's behaviour. Keeping these points in mind, the rest of this chapter discusses and motivates the approach taken for the conversion of the statistical constructs to constructs found within the DATE framework. We shall also informally discuss how the statistical logic semantics are preserved by the conversion. The ideas presented in this chapter will be formally presented in chapter 9.

6.1 Statistical Event Conversion

At the core of the statistical event semantics lie the core notions of automaton synchronisation and information transfer. Consequently, the statistical event semantics can be directly simulated using value passing channels, since such channels can both act as simple synchronisation tools and also pass information over the channels. Since (as discussed in section 3.2) value passing channels are directly reducible to the basic synchronisation channels found within the DATE framework, this implies that channels as specified within

DATEs are sufficiently expressive for the simulation of the statistical event.

A fresh channel is created for each declared statistic declared within the framework, and an associated channel event (as defined in definition 3.2.1) extracting the information from the channel is also defined. Each statistic, upon the completion of the associated statistical update function is to send the new statistic valuation on the channel associated with the statistic. Consequently, all that is required from a particular construct (be it a property or other statistic) so as to trigger on the statistical event of a particular statistic is to listen to the channel event associated with the fresh channel associated to that particular statistic of interest.

6.2 Point Statistic Conversion

As motivated in section 5.2, the three components describing the point statistic are the current statistic valuation, the point of interest and the statistic update. Consequently, simulating the point statistic semantics reduces to simulating all three components using constructs within the DATE framework.

In truth, all three components are directly translatable to such constructs. Since LARVA allows the declaration of additional variables, the current statistic valuation (of a particular type) is stored within an additional variable declaration of the same type as the statistic. The point of interest comprises two components, these being event and boolean expressions, both of which are to be satisfied if the statistic update is to trigger. Moreover, upon the satisfaction of both components the point of interest is said to be triggered, thus executing the statistical update. The statistical update is an executable function responsible for the updating of the statistic valuation. Consequently, the semantics given to the point of interest and statistic update can be jointly simulated using a DATE transition. A transition is described using three components, the event expression which triggers the transition, a boolean condition which is to evaluate to true if the transition is to trigger and also an action which is to execute upon the triggering of a transition.

The following template describes a general approach to the translation of the point statistic to a DATE

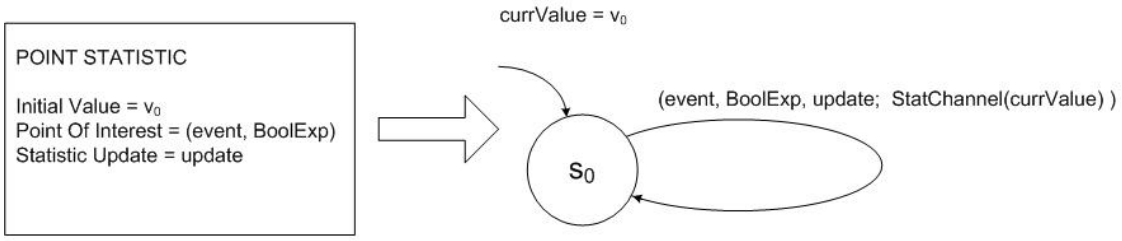


Figure 6.1: Point Statistic Conversion

A point statistic is converted to a one state DATE. The point of interest and statistic update are converted into one transition looping on the initial state. Hence, the transition's event expression and boolean condition are extracted from the statistic's point of interest, whereas the action is set to the statistic update. Variable currValue signifies the statistic's current value, and is initialised to v_0. Note that an additional component (statChannel(currValue)) is added to the action which signifies the sending of the current statistic valuation over the fresh channel associated with the point statistic. This is done so as to simulate the statistical event, since any component listening to that channel receives the new statistic valuation upon the termination of the statistic update. Also note the ordering within the action; the statistic update is executed first and is followed by the sending of the statistic valuation on the associated channel. This is done so as to send the latest statistic valuation on the channel, not the value prior to the statistic update execution.

Note that the resulting DATE lacks any accepting or rejecting states, since statistic collection has no notion of acceptance or rejection, but only of evaluation. Also note the lack of use of other functionality offered by DATEs, such as automaton constructors or clocks. This is because the point statistic semantics lacks any notion of dynamic statistic replication, as well as lacking any reference to time.

6.3 Interval Statistic Conversion

Section 5.3 characterises the notion of an interval statistic through four components, these being the current statistic valuation, the point of interest, the statistic update and the interval. The interval statistic is equivalent to a point statistic over a subset (interval) of the event trace. Consequently, the semantics of the current statistic valuation, point of interest and statistic update are equivalent to those defined for the point statistic. This semantic equivalence permits the same simulation technique as that used for the point statistic for these three components. Therefore, the only component characterising the interval statistic which we need to translate is that of the interval.

Unlike the previous components which are directly translated to a construct defined within LARVA, we shall be taking a different approach toward the simulation of intervals. In fact, the choice of interval type, be it an event, duration or time interval, directly influences the DATE structure which will be emulating the interval statistic semantics. We shall hence be looking at three different conversions, one for each interval statistic with a particular interval type. Let us start from the event interval statistic conversion

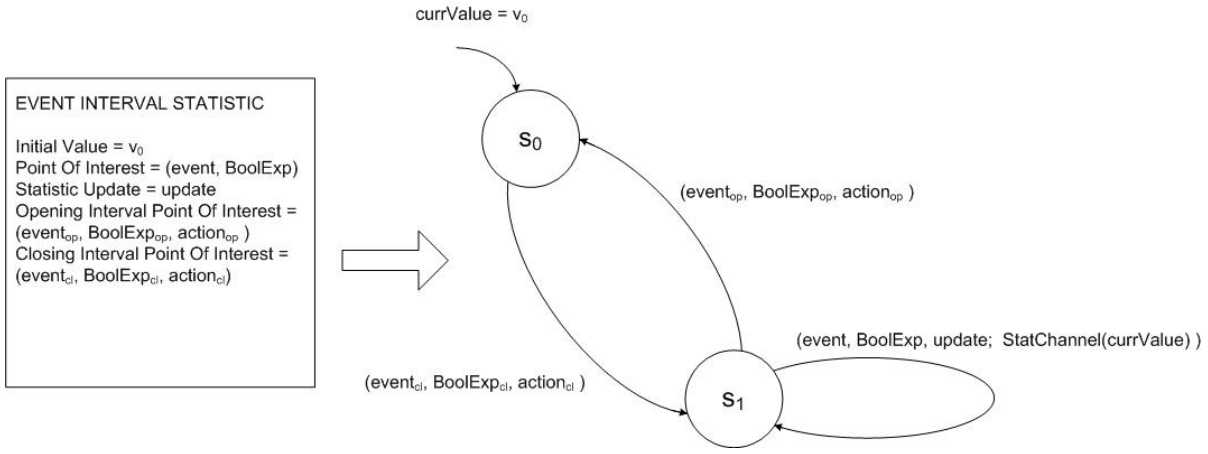


Figure 6.2: Event Interval Statistic Conversion

The event interval statistic is simulated using a two-state DATE. The use of two states (as opposed to one for the point statistic) reflects the fact that any event interval statistic can be in one of two states; either within an interval (s_0) or outside of an interval (s_1). The statistic state reflects what event expressions are of interest to the statistic. If the statistic is outside of an interval then the only event expression of interest is that which triggers the opening of an interval. The event expression triggering the statistical update is of no interest to the statistic if it is still outside the interval, since the event interval statistic semantics allows for the statistical update only when within an interval. Hence, we define an transition from (s_0) to (s_1) characterising the opening interval point of interest, constituting of the opening event expression, the opening boolean condition (both of which must be satisfied in order to trigger the transition) and an action. When the statistic is within an interval, then one of two events can occur which are of interest; the triggering of a statistic update or the closing of the interval. This results in the definition of two transitions from (s_1), one to (s_1) itself in the case of the statistic update triggering and another from (s_1) to (s_0) characterising the closing interval point of interest. Note that once again variable currValue denotes the current statistic valuation, is initialised to (v_0), and is sent upon the appropriate channel after the statistic update execution.

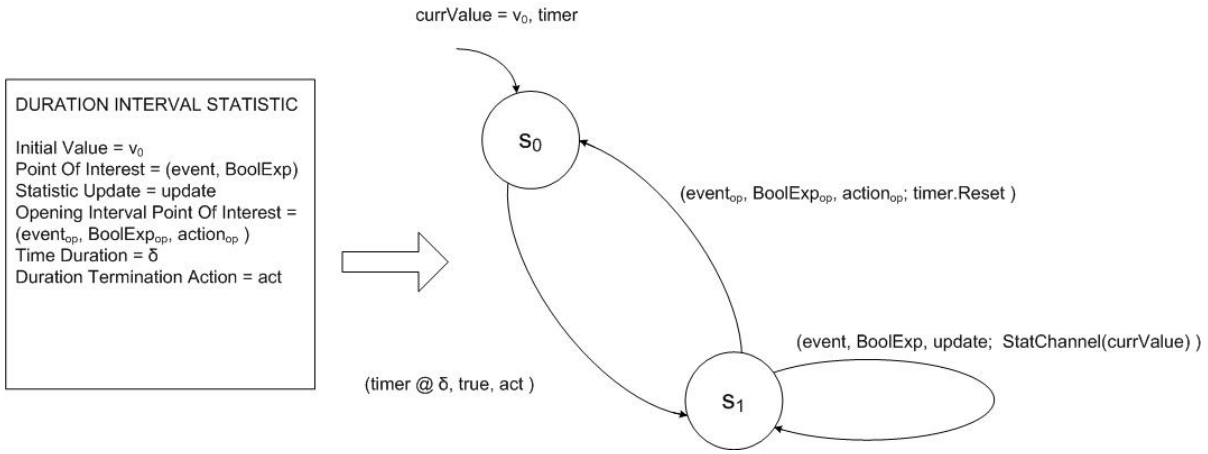


Figure 6.3: Duration Interval Statistic Conversion

The duration interval statistic conversion is almost identical to that used for the event interval statistic, which is analogous to the previous statement that the event and duration intervals are close in nature (the duration interval is reducible to the more general event interval). Hence, the effect of both intervals on the DATE structure representing the interval statistic is almost identical. In fact, the only alterations are the addition of the timer *timer*, and the subsequent use of this timer. The timer is reset upon entering the interval, and triggers the closing of the interval upon elapsing δ time units representing the interval duration. Note that the boolean expression defined within the closing transition is set to true, and the action on the same transition is the defined duration termination action defined for the duration interval.

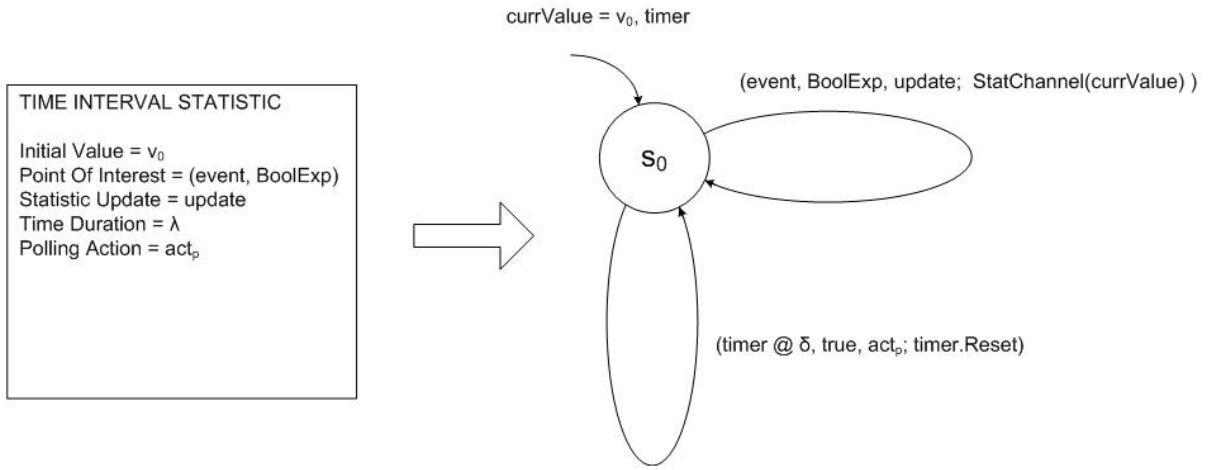


Figure 6.4: Time Interval Statistic Conversion

The time interval statistic conversion uses a somewhat different approach for the conversion of the semantics to DATEs, which is expected due to the dynamic nature of the

interval. The result is a one-state DATE, and mirrors the fact that a time interval statistic always lies within an interval, since it is the interval which moves with time and not the statistic which transitions between intervals. Two transitions are defined, one which is responsible for the simulation of the point of interest and statistic update (as before), and another transition responsible keeping in line the statistic valuation with the time interval semantics. This transition triggers upon the elapsing of δ time units representing the polling interval (as motivated in section 9.1). The polling action act_p is responsible for updating the statistic valuation so that this valuation conforms with the λ interval duration. Moreover, the timer is reset each time this transition is triggered so that the polling mechanism is restarted.

6.4 Conclusions

This chapter informally presented the approach taken for the conversion of our motivated logic into DATEs, thus defining an operational semantics for our logic in the process. The reason for this conversion is in order to provide an executable platform for the execution of the logic, as well as to provide a common ground where both the statistical and runtime verification frameworks can easily communicate. In truth, this presented approach is rather straightforward due to certain design choices taken in chapter5.

The translation of our logic into DATEs offers the clearest picture yet regarding the relationships between our statistical constructs. The most revealing indication lies in the fact that although we have defined the notion of an interval statistic using a possible variety of interval definitions, the choice of interval profoundly impacts the interval statistic behaviour. Moreover, whereas the event and duration intervals are analogous and impact the interval statistic's behaviour in a similar fashion, the time interval presents a completely different semantics with respect to the notion of the interval. This difference is mirrored in the translation of the time interval statistic, which is different than the other two interval statistics. Another observation worth noting is that the presented translations reflect the fact that the interval statistic is an augmentation of the point statistic. This is because the translation of each interval statistic uses the point statistic translation at their core, and integrate the simulation of the appropriate interval.

A crucial observation not lost on us is the fact that this translation between our logic and DATEs implies that DATEs are at least as expressive as our logic, which leads to the question whether our statistical framework is required at all. Nevertheless, as specified by the core aims of this thesis, we require the concise and intuitive specification of complex statistics, which is impossible using only DATEs as shown by the presented translations. This is due to the fact that each statistic is represented by a lengthy and possibly complex set of DATE constructs (using DATEs, clocks and channels). Hence, the concise and intuitive expression of a wide variety of complex statistics can only be

achieved using our defined statistical framework.

7. Language Design

This chapter is an exercise in the design of a domain specific embedded language extending LARVA with statistical capabilities, which we shall henceforth name LarvaStat. This designed language serves as the tool for the concrete expression of both statistics collection at runtime, as well as the augmentation of this functionality upon the runtime verification framework using DATEs, as was extensively motivated in the previous chapter. The core principles kept in mind during the language design, apart from expressivity and intuitive statistic declaration, is the seamless intuitive integration of the added statistical constructs with LARVA. The language will be presented in an evolutionary manner, mainly through the explanation of various BNF snippets. Note that due to LarvaStat being an extension to LARVA, various sections of the LarvaStat script language are in fact extracted from the original LARVA scripting language. Consequently, knowledge of LARVA (refer to [12], section 3) is a prerequisite to the apprehension of LarvaStat.

The chapter starts by giving an overview of the LarvaStat script and the approach taken toward the integration of the statistical framework with LARVA. Following this, each defined statistical construct is given a concrete BNF representation, starting from the point statistic, proceeding to the interval statistic and finishing with the statistical event. Section 7.5 discusses the notion of treating statistics as objects, as well as the associated functionality afforded to these objects. Sections 7.6 and 7.7 conclude the chapter by presenting the BNF for the statistic and interval definitions, representing macro-like structures aiding in reusability within our framework – reusable statistics.

7.1 The LarvaStat Script

The LarvaStat script is an augmentation of LARVA. Hence, core LARVA script runtime verification functionality, as well as the overall LARVA structure and context rules are directly transferred to LarvaStat. Imported LARVA functionality also include dynamic automaton creation, real time capabilities as well as channels. We shall be taking the (previously motivated) approach of treating the statistical framework as an additional layer to LARVA. However, unlike other simpler LARVA script constructs (such as clocks

or channels), the specification of statistics is more complex and adheres to certain ordering rules. Moreover, as specified in section 5.11, each LARVA context will have the functionality of declaring statistics specific to that context. Also, all statistics as well as properties defined within the script are able to communicate through statistical events. The following BNF defines a broad overview of the LarvaStat script.

Context	::=	InvariantsBlock VariablesBlock EventsBlock StatisticsBlock PropertyBlock SubContexts
SubContext	::=	*'FOREACH'* '(' Type Identifier ')' '{' Context '}'
SubContexts	::=	SubContexts SubContext | ϵ
GlobalContext	::=	*'GLOBAL'* SubContext | ϵ

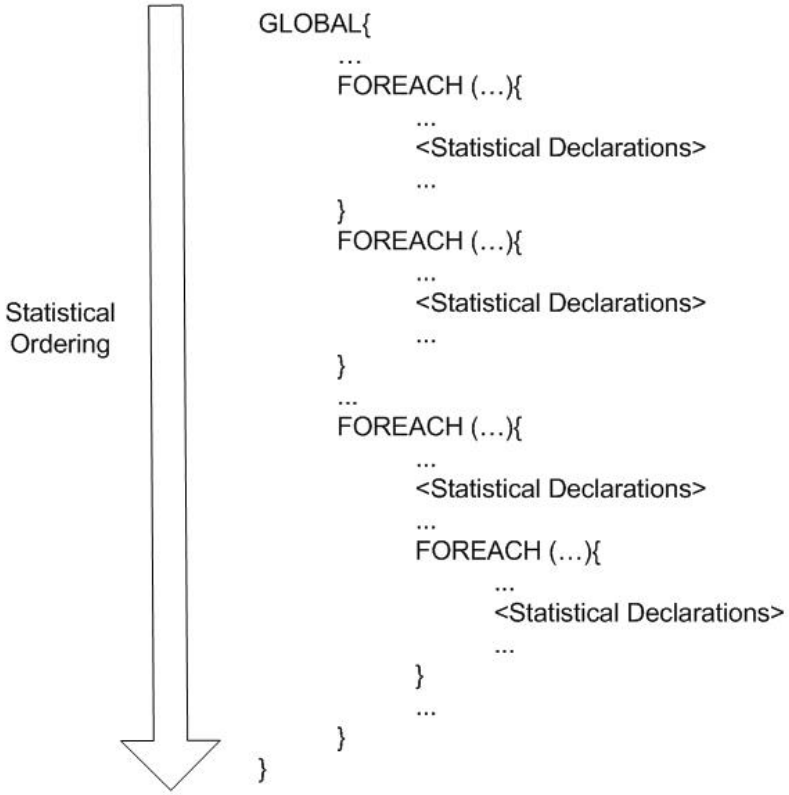
where InvariantsBlock, VariablesBlock, EventsBlock and PropertyBlock all refer to LARVA BNF definitions extracted from [12]. Note that the definition of SubContext, SubContexts and GlobalContext are also identical to those defined for LARVA. The only alteration to the definition of Context as defined for LARVA is the addition of StatisticsBlock representing the statistic declarations defined for each context. The statistic block symbolises the augmentation of LARVA with statistical functionality. We shall not be defining the notions of an invariant, variable, event and property block since we shall make no explicit reference to these LARVA script sections within LarvaStat. In simple terms, the above BNF shows that the LarvaStat script is identical in structure to LARVA, with the addition of statistical declarations at the start of each context. Although not shown in the BNF, other additional LARVA features such as importing, comments and method declarations have also been carried over to LarvaStat. Note that LarvaStat offers guarantees that all statistic updates are executed prior to transitions defined within properties. Hence, all defined properties are guaranteed to have access to up to date statistic valuations for use within property definitions.

The rest of this chapter is mainly focused on describing the defined language for the characterisation of statistics declarations, represented by StatisticsBlock in the above BNF. As has been previously motivated, the description of statistics within our framework comes in two forms; the point and interval statistics.

StatisticsBlock	::=	(PointStatistic StatisticsBlock) | (IntervalStatistic StatisticsBlock) | ϵ

Hence, a block of statistic declarations within a context is defined as a sequence of interleaved point and interval statistic declarations. Note that the sequence with which

statistics are declared within a statistics block is taken to specify the statistic ordering within that context. Hence, if a statistic is defined prior to another within that context, it is taken to be prior to the latter statistic within this ordering. In the case of statistics declared over different contexts the statistical ordering is defined as follows.



which implies that irrespective of context, statistics triggering on the same event expression are evaluated in the order they are defined in the script. Also note that the statistical declarations are also to adhere to the statistical dependency ordering as defined in section 5.10.

Note that statistical declarations within a context are considered to be construct declarations, similar in concept to the declaration of clocks or channels, however more complex due to the nature of statistics collection. Consequently, each statistical declaration can be manipulated (as will be discussed below) by other constructs or sections within the script, as well as java code embedded within other constructs. Moreover, access to statistical declarations obey context rules as defined within LARVA.

A minor issue resulting from the LarvaStat compiler implementation (discussed in the next chapter) affecting the statistical language is the restriction disallowing variable or statistic names starting with the $ symbol, since constructs starting with the $ symbol are

reserved internal constructs created by the compiler. Another note worth remembering is the requirement of all declared statistics having a unique name irrespective of context. This is required for straightforward expression of statistical events, as will be discussed below.

7.2 The Point Statistic

The notion of a point statistic as defined in chapter 5 requires the definition of an initial statistic valuation, a point of interest as well as a statistic update. The point of interest is characterised by the definition of an event expression and a boolean condition. Moreover, we later motivated the functionality of declaring additional variables so as to be able to compute non-incrementally computable statistics. Hence, the following BNF represents the point statistic as expressed within LarvaStat.

PointStatistic	::=	*'POINTSTAT'* Identifier ['[' *'OFF'* ']'] ':' Type '{' PointStatBody '}'
PointStatBody	::=	*'INIT'* '{' JavaCode '}' *'EVENTS'* '{' EventName '}' [*'CONDITION'* '{' Condition '}'] [*'OTHER'* '{' JavaCode '}'] *'UPDATE'* '{' JavaCode '}'

An identifier as well as a statistic type is associated with each point statistic declaration. Whereas the identifier obeys the standard Java naming convention, the associated type broadly refers to any Java reference object type . Hence, whereas types such as Integer, Double or Byte are acceptable, variable types such as int or double are not. In truth, this requirement emerges from the fact that the LarvaStat compiler represents each statistic declaration through an object defined using generics, and since generics can only be used on reference type objects, this constricts us to the use of such types.

Each point statistic is to define the INIT, EVENTS and UPDATE sections, with the option of defining two more, these being the CONDITION and OTHER sections. The INIT section defines initial code to be executed upon statistic initialisation, and usually initialises the statistic valuation. However, additional code can also be optionally added. The events section declares the event name which the statistic triggers on. Note that LarvaStat reuses the rich event semantics defined for LARVA. Hence, any event defined within the current or any super context can be referred to within the pointstat EVENTS section. Also note that an event declaration can extract context from the underlying system. This context (in the form of objects) can be referred to in the CONDITION and UPDATE sections, since both sections are executed upon the event triggering. The

CONDITION section declares a boolean condition which is to evaluate to true if the statistic is to update. The UPDATE section contains java code which is executed upon the triggering of the point of interest, and updates the current statistic valuation. Other necessary variables can be declared within the OTHER section, and can be used for the computation of non-incrementally computable statistics. Hence, variables declared within the OTHER section are accessible to all other sections within the statistic declaration.

Note that whereas each point statistic declaration by default is initialised to on (hence listening for the firing of events and executing the statistic update), the point statistic declaration allows for the functionality of initialising the statistic a switched off state. This can be achieved by adding the [OFF] tag after the statistic identifier declaration. If a statistic is initialised to off, it can be switched on at a later stage using an action on the statistic, as will be discussed below.

The following is an example point statistic counting the number of bad logins for each user login session.

```
GLOBAL
{
    FOREACH (Session s)
    {
        EVENTS
        {
            passwordEvent(ftpdProtocol p,String res) = {p.PASS(arg1)
                        uponReturning (res) } where {s = p.getSession();}
        }

        POINTSTAT BadLoginCount : Integer
        {
            INIT{BadLoginCount.setValue(new Integer(0));}

            EVENTS{passwordEvent}

            CONDITION{res.equals("530 authorization failed. not logged in")}

            UPDATE{ BadLoginCount.setValue(BadLoginCount.getValue() + 1); }
        }
    }
}
```

Point statistic BadLoginCount counts the amount of bad logins for each user login session. Upon the triggering of a password event and the condition verifying that the password attempt has failed, the point statistic valuation is incremented by 1.

7.3 The Interval Statistic

The interval statistic is considered as a point statistic over a limited sequence of the event trace. Consequently, the interval statistic shares a lot in common with the point statistic, with the addition of the interval and other features as specified below.

IntervalStatistic	::=	*'INTERVALSTAT'* Identifier ['[' *'OFF'* ']'] ':' Type '{' IntervalStatBody '}'
IntervalStatBody	::=	*'INIT'* '{' JavaCode '}' *'EVENTS'* '{' EventName '}' [*'CONDITION'* '{' Condition '}'] *'LIST'* '{' 'true' | 'false' '}'] [*'OTHER'* '{' JavaCode '}'] Interval *'UPDATE'* ['[' JavaCode ']'] '{' JavaCode '}'

The interval statistic augments the point statistic definition with the addition of the timestamped object list, the interval declaration as well as the addition of the exclusive update. The remainder of the defined sections remain identical in meaning and scope as their point statistic equivalent.

The timestamped object list is an additional feature defined for interval statistics which aids in the definition of certain interval statistics. In fact, this list is meant mostly for use in conjunction with a time interval statistic due to the extensive functionality defined on the list with respect to time. In essence, the timestamped object list is a list data structure with the additional notion of a timestamp added to each object within the list. The addition of this timestamp allows for the expression of interesting time-based operations. The timestamped object list accepts objects of type TimestampedObject, which encapsulates an object (of the same type as the interval statistic) with an additional timestamp set upon object creation. The following functionality is defined on the TimestampedObject

- getTimestamp() — Returns the timestamp associated with that object.
- getObject() and setObject() — the getter and setter for the encapsulated object of the same type as the declared interval statistic.
- compareTime(Long time) — Returns -1 if argument *time* is older than the object's timestamp, 0 if both are equal or 1 if the timestamp is older than *time*.
- compareTime(TimestampedObject obj) — Overloads the previous method by allowing the comparison of two TimestampedObject objects. The result is analogous to the previous method.

Each interval statistic has the option of declaring a list of type TimestampedObjectList through the LIST section, whereby setting the LIST section to true implies that the interval statistic object (discussed later) will implicitly contain an instance of the list. Of crucial importance is the declaration of the operations defined on this list. The following functionality is defined for the TimestampedObjectList

- add(Type obj) — Adds object *obj* of type *Type* (must be the same type as that declared for the interval statistic declaration.
- contains(Type obj) — Checks if the list contains *obj*.
- Type get(int index) — Returns the object in the list found at location *index*.
- TimestampedObject getTimestampedObject(int index) — Returns the timestamped object in the list found at location *index*.
- clearList() — clears the list of all its contents.
- size() — returns the list size.
- ArrayList getContents() — returns the list contents in the form of an arraylist of objects (thus removing the timestamp).
- updateList(long timeDuration) — Updates the list contents, discarding any object within the list which is 'older' than the specified *timeDuration* argument.

Although most are standard list operations, what is perhaps most interesting are the semantics given to operation updateList, since the semantics given to the updateList operator fits in perfectly with the notion of the polling mechanism periodically updating the time interval statistic. In truth, the definition of the TimestampedObjectList and the associated real time functionality is intended as a mechanism which simplifies the process of executing the complex dynamic semantics defined for the time interval. Using such a list, statistical values can be saved within the list and automatically be discarded when items within the list are outdated, which fits in perfectly with time interval semantics. However note that the TimestampedObjectList is not exclusive to the time interval statistic, and can be used in conjunction with any other interval statistic. Also note that if a time interval statistic declares the requirement of a TimestampedObjectList (through the LIST section), this implies that the same list will be implicitly updated, using the time duration defined for the time interval, within the underlying polling mechanism in order to keep the list updated. All this underlying logic is hidden from the user for simplicity and clarity. It is for this reason that we believe that such tools greatly aid in the definition of certain statistics.

The exclusive update is a syntactic sugar added to the UPDATE section which allows the definition of two update update code snippets. Note that the definition of the exclusive update is permitted only in conjunction with the definition of time interval statistics (hence use in conjunction with other interval statistic types is disallowed). The addition of the exclusive update derives from the choice of polling as the execution mechanism for the time interval. Since the

interval updates continuously through polling, we require two update functions; one which is executed upon the triggering of the point of interest (the exclusive update), and another which executes upon upon triggering of the point of interest as well as during polling. Hence, the code placed in between square brackets is considered to be the exclusive update, whereas the code placed in between curly brackets is the code executed during polling. In truth the second update can be encoded within the time interval declaration (defined below). However, the separation into two updates aids in the intuitive design of the time interval statistic updating mechanism.

Note that similarly to the point statistic, the interval statistic can also be initialised to off and switched on later by another construct defined within the script. This is achieved by executing an action on the statistic as will be defined below.

7.3.1 Interval Representation

The third feature exclusive to the interval statistic is the definition of an interval. This feature is in fact the crucial feature which distinguishes the interval from the point statistic. As motivated in section 5.4, the statistical framework defines three interval types, these being the event, duration and time intervals. Moreover, the building block of both the event and duration interval is the interval point of interest, of which there are two; the opening and closing interval point of interest.

OpenIntervalPointOfInterest	::=	*'OPEN'* '[' EventName '\' [Condition] '\' [Action] ']' *'WHERE'* '{' JavaCode '}'
CloseIntervalPointOfInterest	::=	*'CLOSE'* '[' EventName '\' [Condition] '\' [Action] ']' *'WHERE'* '{' JavaCode '}'

where each interval point of interest is defined through the event name (declared in the event block), optionally a boolean condition (if missing assumed to be true) and also an optional action which stands for code which is to be executed upon triggering of the interval point of interest. Note that the first two components make up the point of interest whereas the third component is the action as defined in section 5.4. Both the opening and the closing interval points of interest have identical semantics. We will for now ignore the WHERE clause defined for both points of interest since it relates to an advanced feature related to context. However, note that the code defined within the WHERE clause can refer to context extracted by the event it is triggered on. We will return to this issue in depth later when defining overlapping intervals through context.

The three interval types are represented below

```
EventInterval     ::=  OpenIntervalPointOfInterest CloseIntervalPointOfInterest
DurationInterval  ::=  OpenIntervalPointOfInterest
                       'DURATION' '[' PositiveInteger '\' [ Action ] ']'
TimeInterval      ::=  'TIME' [ '(' 'OnUpdate' | 'OnChange' ')' ]
                       '[' PositiveInteger '\' [ Action ] ']'
```

As motivated in section 5.4, an event interval is characterised through an open and close interval point of interest, a duration interval is defined through an opening interval point of interest, a duration of time (denoting the interval duration) and an action executed upon closing of the interval, and the time interval is defined through an integer denoting the time window duration and also an action periodically executed for the purpose of polling. Note that the declared duration for both the duration and time interval represents a duration in seconds. Also note that the time interval statistic internally updates its valuation (through the execution of the polling action) every 0.5s, as will be motivated in the next chapter.

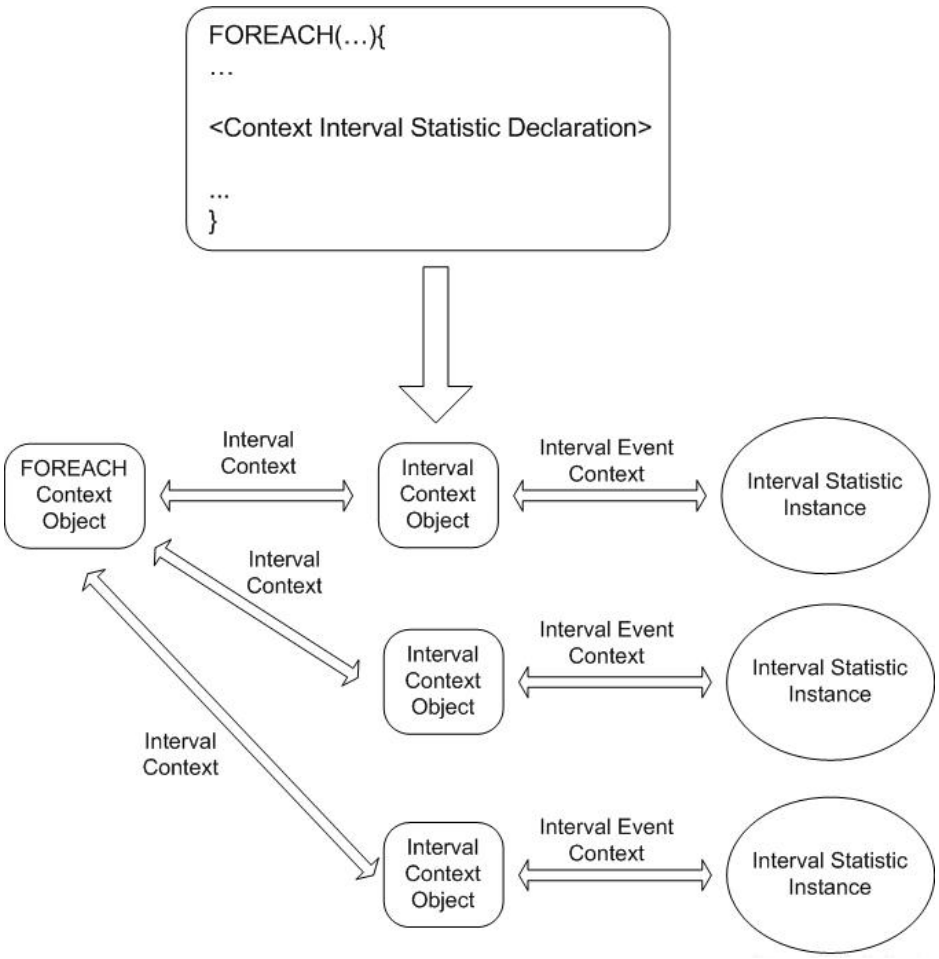
Of note is the optional declaration of the Time Interval with an 'OnUpdate' or 'OnChange' tag. This feature relates to the statistical event construct discussed in the next section, whereby additional control is given to the user on when to trigger the statistical event. Declaring the time interval with an 'OnUpdate' tag triggers a statistical event each time the time interval statistic is polled, whereas declaring the statistic 'OnChange' triggers a statistical event each time the associated point of interest is triggered. By default, the time interval point statistic assumes that the statistic is defined as 'OnChange'.

```
IntervalDecl  ::=  EventInterval | DurationInterval | TimeInterval
Interval      ::=  'INTERVAL' [ '[' 'CONTEXT' '(' Type Identifier ')' ]
                   '{' IntervalDecl '}'
                   [ 'OBJECTCONTEXT' '' JavaCode '' ]
```

An interval declaration is either an event, duration or time interval. Of crucial importance is the additional functionality of encapsulating the interval declaration within a context in the form of an object (similar in concept to the binding of an automaton with a context as defined in LARVA). The addition of context for intervals is the presented solution for the issue of overlapping intervals defined in section 5.4. Hence, the issue of context is only relevant to interval statistics which allow for overlapping intervals. Context declaration can be omitted entirely for statistics with non-overlapping intervals. Note that since the time interval semantics disallows for overlapping intervals (as previously motivated), the feature of context used in conjunction with time interval statistics is disallowed.

Given the declaration of a statistic whose interval is associated with a context, this implies that the interval statistic allows for overlapping intervals. Hence, the defined context serves as a

mechanism with which to distinguish between intervals. A new instance of the interval statistic is generated upon the creation of each new context which the interval definition is bound to. Using this context, we can determine which interval is responsible for the triggering of the interval points of interest. This logic implies that the act of distinguishing between intervals is based on two levels; distinguishing between contexts and distinguishing between the triggering of interval points of interest, as explained below



In general, each statistic declaration is embedded within other contexts defined by FOREACH statements. Hence, the interval context is responsible for linking the interval context object with the rest of the super contexts. If an interval is nested within the n^{th} context, then the interval context requires the definition of all n super contexts. The notion of the interval context is defined within the OBJECTCONTEXT parameter in the form of Java code. Note that the declaration of interval context is omitted in the case of a context interval statistic being defined in the GLOBAL context. The reason for this is that the GLOBAL context by definition is global lacking any object defining its context, and hence no link can be specified from the interval context object to the GLOBAL context. The interval event context distinguishes between interval contexts. Since the defined interval context object is bound to an interval

statistic instance, using the interval event context we can distinguish which context object is responsible for the triggering of an interval point of interest, and hence which interval statistic is affected. The interval event context is defined within the script through the definition of the WHERE parameters on both the interval points of interest defined above. Note that the object representing the interval context can be referred to in the INIT, CONDITION, OTHER and UPDATE sections, as well as in the declarations of interval points of interest and WHERE clauses.

Whereas the language defines sufficient functionality for the expression of interval context for the opening and closing of interval points of interest through the WHERE clause (as specified above), there exists no such direct functionality for the expression of interval context for the statistic point of interest (which triggers the statistic update) should it be required. In other words, certain interval statistics admitting overlapping intervals occasionally require a manner with which to selectively choose (between currently open intervals) which statistic instances are allowed execution of the statistic update. Such context can be defined through the use of the CONDITION section, whereby the defined condition acts as a context filter, as will be seen below.

The following example defines a context interval statistic (hence allowing for overlapping intervals) which collects the total amount of bytes downloaded per connection.

```
GLOBAL
{
    EVENTS
    {
        sendInfo(OutputStream out) = {call out.write(arg1) }

        downloadStarting(Connection c) = {call c.eventDownloadFilePre()}
        downloadComplete(Connection c) = {call p.eventDownloadFilePost()}
    }

    ...
    FOREACH (Session s)
    {
        ...
        INTERVALSTAT BytesDownloadedCountPerConn : Integer
        {
            INIT{BytesDownloadedCountPerConn.setValue(new Integer(0));}

            EVENTS{sendInfo}

            CONDITION{ out == conn.bOut }

            LIST{false}
```

```
            INTERVAL [CONTEXT (Connection conn)]{
                OPEN[downloadStarting \ \ ] WHERE {c == conn}
                CLOSE[downloadComplete \ \ ] WHERE {c == conn}
            }   OBJECTCONTEXT { s = conn.getSession(); }

            UPDATE{ BytesDownloadedCountPerConn.setValue(
                          BytesDownloadedCountPerConn.getValue()
                                                  + conn.bufferSize); }
        }
        ...
    }
}
```

Each connection created within the underlying system is bound to an instance of statistic BytesDownloadedCountPerConn. Note how the OBJECTCONTEXT parameter links the connection object with the rest of the super contexts, in this case the object representing the user login session. Also note that variable c is extracted from the system by event sendInfo(), and refers to the connection object which triggered the event. Each time a new connection is created, this automatically triggers the creation of a new statistic instance. As the system executes, a user initiates the start of a download. Since this triggers the opening interval point of interest, we require to distinguish which connection is responsible for the download trigger, so as to be able to update the required statistic accordingly. This is done through the WHERE clause added to each point of interest, whereby the connection triggering the download start must be equal (as in the same object instance) to the same object bound to the interval statistic. With the issue of overlapping intervals solved, updating the interval statistic simply entails listening for the sending of a byte event, and after ensuring that the byte has been sent by the same connection as that defined as the interval context object (through the context filter), updating the statistic count by the connection buffer size.

Note that due to a section ordering limitation with LARVA (where any FOREACH section must come after the property definitions), we require to slightly alter the notion of statistical ordering as defined previously. Given a context defining a set of statistics, a subset of which are context interval statistics, then this subset is placed (within the ordering) post the remainder of the statistic declarations, but prior to the statistics within the succeeding contexts. Within this subset, the ordering is implied from the declaration ordering within the script. Also note that we require the interval context object to have a defined constructor, since the creation of the interval statistic is bound to the creation of the interval context object.

7.4 The Statistical Event

A statistical event is triggered upon the execution completion of a statistic update (or in the case of a time interval statistic, possibly upon the statistic being polled) containing the statistical valuation upon the completion of the update execution. Unlike other events declared within LARVA, the statistical event is implicitly defined for each statistic defined within the LarvaStat

script. Hence, no explicit event declaration is required for the declaration of a statistical event.

Any construct wishing to listen to a statistical event, be it another statistic or a property definition, can do so by simply registering with the statistical event associated with the particular statistic. Given a statistic declaration identified through the name *stat*, the statistical event is declared under the name *stat* + "_Event". Moreover, analogously to other event declarations returning context information, the statistical event returns information encoding the value sent on the statistical event. This value is also implicitly defined and is declared under the name *stat* + "_Value" having the same type as the statistic declaration, and can be accessed by associated actions executing upon the triggering of a statistical event. Assuming a statistic construct declaration AverageDownloadCount, the following are two example constructs making use of the associated statistical event.

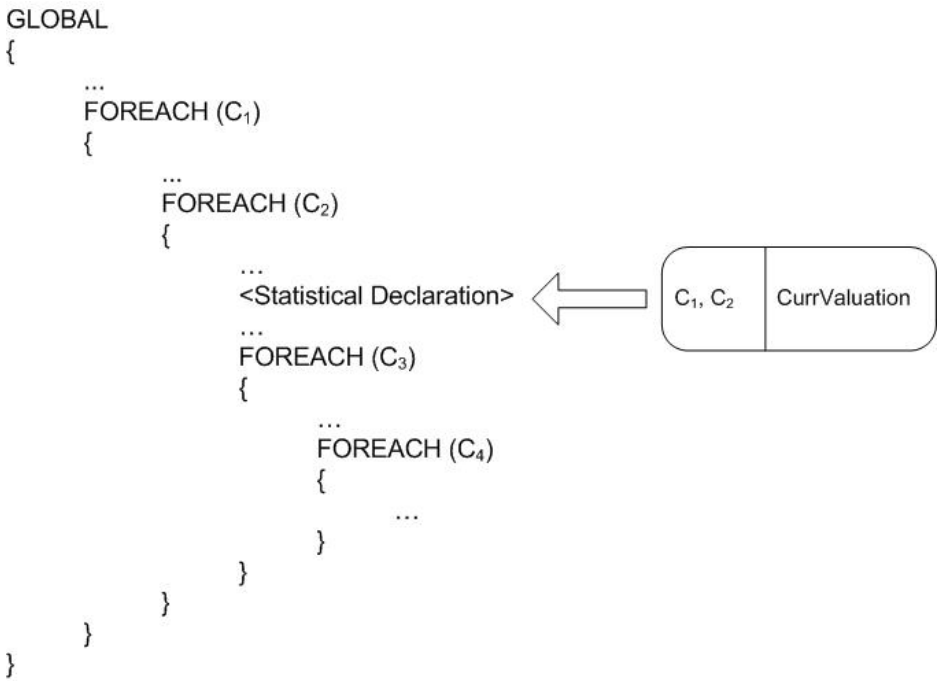
```
start -> start[AverageDownloadCount_Event \ \
                          System.out.println(AverageDownloadCount_Value); ]

...

POINTSTAT maxValue : Integer
{
    INIT{ maxValue.setValue(new Integer(0)); }

    EVENTS{ AverageDownloadCount_Event }

    UPDATE{ if (AverageDownloadCount_Value > maxValue.getValue()
                        maxValue.setValue(AverageDownloadCount_Value); }
}
```

The first example is a transition listening on AverageDownloadCount's statistical event, and outputting the value sent on this statistical event. The second example is a point statistic declaration evaluating the largest average download count. The statistic defines its point of interest by listening to AverageDownloadCount's statistical event (hence defining a two-layered statistic declaration). Moreover, the defined statistic update checks if the value sent over the statistical event is larger than the current statistic valuation, and if so sets the statistic valuation to this value.

Since certain interval statistics may not admit a direct mapping between the closing of its interval and an event (especially in the case of duration interval statistics), we shall extend the statistical event so as to generate an event upon the closing of an interval. Hence, apart from the generation of statistical events upon execution of the statistical update, an additional event is generated transmitting the statistical valuation upon the closing of the interval. This additional event is applicable for event and duration interval statistics (the time interval never closes, but updates), since it is these two interval statistics which admit intervals which occasionally close. This new statistical event is implicitly defined, and can be accessed by any other construct analogously to the previous statistical event. Given interval statistic *stat*, the statistical event

upon the interval closing is declared under the name of *stat* + "_EventComplete", whereas the valuation sent over this event can be accessed by the name *stat* + "_FinalValue".

The issue of context is hidden from the statistical event, which implies that no explicit context declaration is required when listening to a statistical event. This does not however imply that the statistical event does not obey certain rules when handling context. Each statistical event implicitly determines its context as explained below



The statistical event associated within the declared statistic implies two contexts, the context it is declared in and its super context. In general, any construct listening on a statistical event uses as much context information as is required, or available. Hence, a construct declared within the first FOREACH statement listening to the statistical event would implicitly use only C_1 to determine its context. A construct declared within the second FOREACH statement would use all context information implied by the statistical event, whereas all constructs defined in sub contexts can only use context information C_1 and C_2. The statistical event for constructs embedded within such sub contexts hence results in the event acting as a broadcast event to all sub contexts.

7.5 The Statistic Object

Statistic declarations within LarvaStat are considered to be construct declarations, conceptualised analogously to clock or channel declarations. Hence, we shall be taking a similar approach for the representation of statistics as that taken for such constructs, whose representation is achieved through object instances and associated functionality is implemented through

method declarations. This leads us to the creation of the statistical object, responsible for storing the statistic valuation as well as giving the user the required functionality to execute operations on the statistic (thus allowing for the implementation of actions on statistics, as motivated in section 5.5).

We require a class structure which enables us to implement two distinct, yet related notions representing the point and interval statistic.

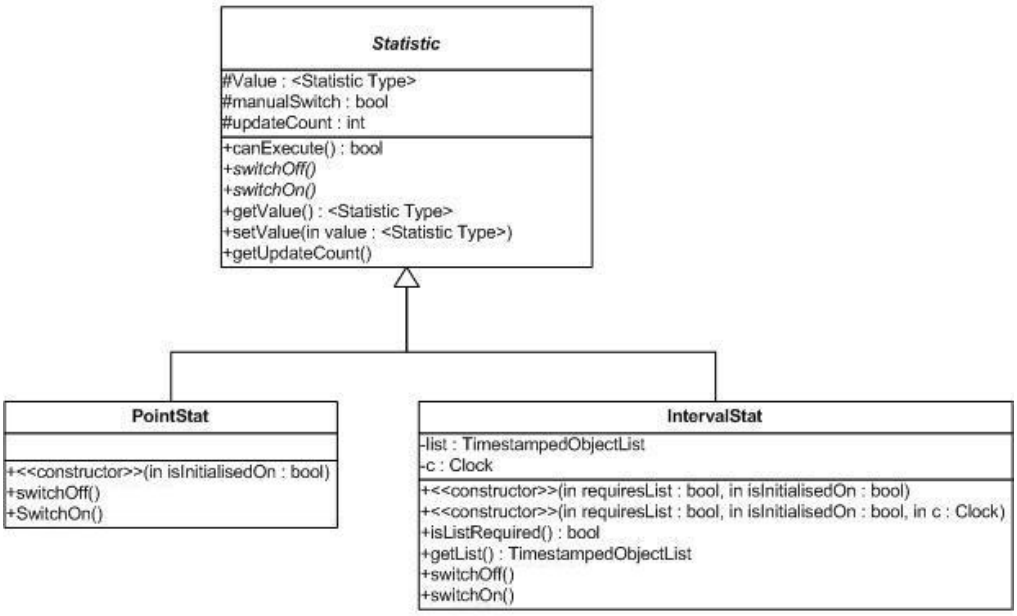


Figure 7.1: UML Class Representation

Each statistic declaration is represented by an object instance of the appropriate type, and is referred to by the same name as that given to the statistic declaration. Moreover, using standard object oriented mechanisms such as typecasting and polymorphism, object instances representing statistics can be treated indiscriminately as an abstract Statistic object. Note that the variable type storing the statistic valuation dynamically changes (according to the defined statistic type) without affecting the defined functionality. This variable type must be a reference type object. The Statistic abstract class encapsulates the similarities present in both statistic types, and defines the following functionality

- canExecute() — Returns whether the statistic is currently manually switched on — hence does not factor whether the statistic is within an interval.
- switchOff() — An abstract method specifying that a statistic can be switched off.
- switchOn() — An abstract method specifying that a statistic can be switched on.
- getValue() — Method defining the retrieval of the current statistic valuation.

- setValue() — Method allowing for the alteration of the current statistic valuation.
- getUpdateCount() — returns an internal counter specifying the amount of times the statistic valuation has been updated.

The PointStat class represents point statistic construct declarations, and implements the Statistic abstract class. Hence, the point statistic inherits the functionality previously defined for the Statistic class, and also implements the switchOn() and switchOff() abstract method declarations. Note that the defined constructor takes one parameter which specifies whether the point statistic is initialised to on.

The IntervalStat class represents interval statistic declarations, and also extends the Statistic abstract class, inheriting its functionality and implementing the declared abstract methods. Moreover, the IntervalStat class extends this functionality by storing a TimestampedObjectList (discussed previously) list which can optionally be used in conjunction with the interval statistic. Hence, the interval statistic extends the Statistic functionality through the following two methods

- isListRequired() — returns whether the list is instantiated.
- getList() — returns the TimestampedObjectList list object.

As has been previously discussed, the TimestampedObjectList list offers rich functionality with respect to interval statistics. Two overloaded constructors are defined, one which takes two parameters specifying whether the list functionality is required and whether the statistic is initialised to on, whereas the other constructor additionally takes as a parameter a Clock instance. As previously discussed, certain interval statistics additionally use the functionality of a clock which affects the statistic's execution. Hence, pausing or restarting an interval statistic also affects the associated clock's state accordingly.

Although the currently defined functionality on both statistic classes could technically be implemented with the interval statistic being a subset of the point statistic, we want to convey the fact the interval statistic is not a specialisation of the point statistic. In other words, the semantics defined for both statistic constructs does not imply that the point statistic is a generalisation of the interval statistic. Instead, the interval statistic extends and also substantially modifies the notion of statistics collection when compared to the point statistic.

In short, each statistic construct declaration is represented by an instance of the appropriate class. The functionality defined for each class defines access to the current statistic valuation, as well as substantially simplifies control over the process of statistics collection. Note that all statistical objects obey context rules as any other construct defined in LARVA. Hence, each statistic object is accessible to all sub contexts, as well as the same context the statistic is defined in. Note how all actions on statistics (motivated in section 5.5) are implemented through the statistical object.

7.6 The Statistic Definition

The specification of statistics often results in the reuse of logic. A set of statistics may for example keep count of the number of occurrences of different events (hence reusing the logic defining the counting of an event), whereas another statistics collection may calculate the average value of certain value sequences generated at runtime (hence reusing the algorithm which evaluates average of a set of values in an online fashion). In practice, this reuse of logic often results in specification redundancy especially in scripts of considerable size, since a significant amount of statistics are analogous in nature with only minor alterations, mostly regarding issues such as which event to listen to.

We therefore require a mechanism which allows for the definition of reusable statistics. This required mechanism is aided by what we broadly perceive as two distinct natures characterising a statistic declaration (as specified previously). These two notions are defined as

- **Statistic behaviour characterisation** — Specification of the logic defining the statistic behaviour; hence encapsulating information such as the initial statistical valuation, the statistical update function and other variables required for the computation of non-incrementally computable statistics.
- **Specification linking the statistic with the system** — With the statistic behaviour characterised, we require the specification linking the characterised statistic with the underlying system. Such information includes the specification of the point of interest, and in the case of interval statistics also the specification of interval points of interest.

This separation of the statistical declaration into two separate notions allows for the straightforward definition of reusable statistics. This is achieved by the creation of macro-like structures, which we shall henceforth refer to as statistic definitions, with which to specify statistic behaviour separately from the LarvaStat script. Consequently, statistic definitions are instantiated throughout the script, and upon doing so can specify the appropriate link with the system. This approach allows for the unique specification of logic governing the statistic, and multiple instantiations of this logic as required, which is what we require.

We require a language for the specification of point statistic and interval statistic definitions. As motivated above, such statistic definitions specify a subset of the required information characterising the statistic, more specifically information relating to statistic behaviour.

PointStatisticDef	::=	*'POINTSTAT'* Identifier ':' Type '{' PointStatDefBody '}'
PointStatDefBody	::=	*'INIT'* '{' JavaCode '}' [*'OTHER'* '{' JavaCode '}'] *'UPDATE'* '{' JavaCode '}'
IntervalStatisticDef	::=	*'INTERVALSTAT'* Identifier ':' Type '{' IntervalStatDefBody '}'
IntervalStatDefBody	::=	*'INIT'* '{' JavaCode '}' *'LIST'* '{' 'true' | 'false' '}'] [*'OTHER'* '{' JavaCode '}'] *'UPDATE'* ['[' JavaCode ']'] '{' JavaCode '}'

The specified language for both statistic definitions is a subset of the previously specified language describing statistic constructs, specifying only the required information for the specification of the statistic behaviour. Hence, implementing a statistical definition requires no additional knowledge to that defined for the specification of statistics. Note that the identifier defined for each statistic definition is used within the LarvaStat script in order to identify which statistic definition is instantiated, as shown later.

The specification of statistics frequently refer to their own statistical object especially for statistic initialisation or updating. This is done by referring to the statistic object using the same name as the statistic. This poses a limitation to statistic definitions, since statistic definitions are not yet instantiated during their definition. Hence given statistic definition *statDeclf*, we solve this limitation using the notion of tags, whereby the object representing the statistic construct instantiating the statistical definition can be referred to using the '#' + *statDef* + '#' tag. Upon instantiation of a statistic definition, the tag is resolved to the refer to the appropriate statistical object.

We take the notion of tags further, by allowing statistic definitions to refer to the tags of other statistic definitions. This allows for the definition of statistics which are dependent on other statistics, such as a statistic evaluating the average which is hence dependent on the statistic keeping count. Given a statistic definition which refers to the tag of another, it is important to ensure that if the former statistic is instantiated, then so must the latter. Moreover, the former statistic must be instantiated after (within the LarvaStat script) the latter in order to abide by the statistic ordering. Statistic instantiation ordering can also be defined across contexts, as long as the statistical ordering is abided to.

All statistic definitions are packaged within a script separately from the LarvaStat script for better modularity and reusability. The following is an example package specifying two statistic definitions.

```
PACKAGE StatDefPackage
```

```
{
    POINTSTAT EventCount : Integer
    {
        INIT{ #EventCount#.setValue(new Integer(0)); }

        UPDATE{ #EventCount#.setValue(#EventCount#.getValue() + 1); }
    }

    POINTSTAT MaxValue : Integer
    {
        INIT{ #MaxValue#.setValue(new Integer(0)); }

        UPDATE{ if (#MaxValue# < #EventCount#)
                    #MaxValue#.setValue(#EventCount#.getValue()); }
    }
}
```

Each script encapsulating the specification of a set of statistic definitions is to be declared using the 'PACKAGE' reserved word, followed by an identifier associated with the package. This name is part of the statistic definition's fully qualified name, as discussed below. Note that the latter statistic definition specified uses the tag of the former statistic definition. This implies that if statistic definition MaxValue is instantiated within a LarvaStat script, then so must EventCount at a location prior to MaxValue within the statistical ordering.

With the specification of the behaviour of statistic declarations modularised into packages, we next require a method how to instantiate these statistic definitions. Firstly, all required packages are imported to the LarvaStat script as follows

```
IMPORT
{
    ...
}
STATDEFINITIONS
{
    StatDefPackage.txt;
    StatDefPackage2.txt;
    ...
}
GLOBAL
{
    ...
}
```

Each package filename which we wish to import is separated by a semicolon. Importing packages allows for statistic declarations within any context to instantiate definitions located

within the imported packages. Instantiating a statistic definition involves a statistic declaration to refer to the definition by its fully qualified name, and specifying the necessary information linking the statistic with the system. We will hence augment the BNF defining the point and interval statistics as to allow for the instantiation of statistic definitions.

PointStatistic ::= *'POINTSTAT'* Identifier ':' Type '{' PointStatBody '}' |
'=' StatDefinitionName '[' EventName '\' [Condition] ']'

IntervalStatistic ::= *'INTERVALSTAT'* Identifier ':' Type '{' IntervalStatBody '}' |
'=' StatDefinitionName '[' EventName '\' [Condition] '\' Interval ']'

where StatDefinitionName refers to the fully qualified name of a statistic definition imported within a package. In general, the fully qualified name is defined as ⟨ PackageName ⟩ '.' ⟨ StatisticDefinitionName ⟩. Hence, in the case of statistic definition EventCount previously discussed, the fully qualified name is StatDefPackage.EventCount. Note how both the point and interval statistic allow for the instantiation of a statistic definition by referring to an imported statistic definition and specifying the required information linking the statistic with the system. In truth, the information required upon instantiation can be considered to be the remainder of the statistic declaration not allowed within a statistic definition. The following are a few example statistic declarations instantiating previously defined statistic definitions.

```
...

POINTSTAT DownloadCount = StatDeclfPackage.EventCount[downloadComplete() \ ]

POINTSTAT MaxDownloadCount = StatDefPackage.MaxValue[downloadComplete() \ ]

POINTSTAT UploadCount = StatDefPackage.EventCount[uploadComplete() \ ]

...
```

All statistic definitions can be instantiated numerous times, which achieves the main objective of reusability. Note how EventCount is instantiated twice, once to count the number of downloads and again to count the number of uploads. This functionality results in the concept of reusable statistics, which aids in the conciseness of our language. However note that statistic declarations are not mandatory, and can be avoided entirely. Also, MaxValue is instantiated after the first EventCount since MaxValue is defined using EventCount's tag. We require this ordering for the correct evaluation of statistic declarations DownloadCount and MaxDownloadCount.

On a final note we would like to remark that although reusable statistics are a helpful feature which aids in conciseness, not all statistic declarations can and indeed should be defined using statistic definitions. In fact, statistics whose behaviour is dependent on the system (such as the statistic update depending on context extracted from the underlying system) should

not be defined using statistic definitions. This distinction gives rise to the further segregation of statistics, that is statistics with context independent behaviour and statistics with context dependent behaviour.

7.7 The Interval Definition

We shall extend the notion of reusable statistics through the specification of interval statistics to the notion of intervals, thus allowing for the reuse of interval specifications. We shall henceforth refer to the specification of reusable intervals as interval definitions. However, unlike statistic declarations which are independent of any system implementation, the notion of an interval is inherently bound to the underlying system whose subtrace it is defining. Hence, whereas statistics definitions are defined separately from the LarvaStat script and modularized into packages, interval definitions are defined within the same script. This implies that the reuse of interval definitions is only applicable to intervals defined within the same script.

IntervalDef ::= *'INTERVAL'* Identifier '=' ['[' *'CONTEXT'* '(' Type Identifier ')']
'{' IntervalDecl '}'

Where the identifier associated with each interval definition is used as a reference for interval instantiation. Note how although the interval declaration allows for the definition of a context bound to the interval, it does not allow for the declaration of the interval context parameter (specified through OBJECTCONTEXT). This is so since the context interval essentially acts as the link between the interval context and the FOREACH context it is defined within, and hence implies that the OBJECTCONTEXT declaration is sensitive to the context where the interval definition is instantiated.

A simplified version of the tag notion is also applied to the interval definition, whereby using the '#statistic#' tag we can refer to the statistical object which the interval definition is eventually instantiated within. This allows for interval definitions to initialise or finalise the interval statistical valuation upon opening\ closing of the interval. Note that the notion of tag dependencies as defined for statistical definitions has not been applied to interval definitions, since no notion of interval dependencies has been recognised. However, since interval definitions are specified within the same script as that where they are instantiated, we can still refer to statistics declared within the script. When doing so we must however make sure that the statistic instantiating the interval has access to the statistic being referred to according to the context rules.

As has been previously motivated, the specification of interval definitions is placed within the same LarvaStat script where they are instantiated as shown below

```
IMPORT
{
    ...
```

```
}
STATDEFINITIONS
{
    ...
}
INTERVALS
{
    INTERVAL openDownloadConnection = [CONTEXT (Connection conn)] {
        OPEN[downloadStarting \ \ #statistic#.setValue(new Integer(0)); ]
                                                          WHERE {c == conn}
        CLOSE[downloadComplete \ \ ] WHERE {c == conn}
    }
    ...
}
GLOBAL
{
    ...
}
```

Interval definitions can be specified within the INTERVALS section, which is to be placed prior to the GLOBAL section. Note how interval definition openDownloadConnection initialises the statistical valuation of the statistic which eventually instantiates the interval definition upon the interval opening.

Any interval defined within the script can instantiate the specified interval declarations. Hence, both interval declarations upon the statistic definition instantiation as well as intervals defined within interval statistic constructs can instantiate such interval definitions. We will therefore augment the defined interval BNF as follows

Interval ::= *'INTERVAL'* (['[' *'CONTEXT'* '(' Type Identifier ')']
'{' IntervalDecl '}' [*'OBJECTCONTEXT'* '' JavaCode '']) |
'=' IntervalName ['[' 'OBJECTCONTEXT' '{' JavaCode '}' ']']

Where IntervalName refers to the name given to the interval definition. Note how the interval context is defined (through the definition of the OBJECTCONTEXT parameter) upon interval definition instantiation. This serves as the link between the defined interval context object and the context the statistic is defined within. Conversely, defining an interval context for an interval definition lacking context results in the defined interval context being ignored. The following are a few example constructs instantiating the previously discussed interval definition.

```
INTERVALSTAT BytesDownloadedCountPerUser : Integer
{
```

```
    INIT{BytesDownloadedCountPerUser.setValue(new Integer(0));}

    EVENTS{sendInfo}

    LIST{false}

    INTERVAL = openDownloadConnection [
                        OBJECTCONTEXT { s = conn.getSession(); } ]

    UPDATE{ if (out == conn.bOut)
         BytesDownloadedCountPerUser.setValue(
                 BytesDownloadedCountPerUser.getValue()
                                              + conn.bufferSize); }
}
...
INTERVALSTAT AverageBytesDownloaded = StatDefPackage2.AverageVal
      [ sendInfo \ \ INTERVAL = openDownloadConnection
                      [ OBJECTCONTEXT { s = conn.getSession(); } ]  ]
```

Whereas the first example instantiates interval definition openDownloadConnection within an interval statistic declaration, the second example instantiates the same interval definition within the instantiation of a statistic definition. Hence, since interval definitions can be instantiated multiple times throughout the script, this implies the achievement of interval reusability, which is what we require. Note that had interval statistic AverageBytesDownloaded been instantiated within the GLOBAL context, the interval context (defined through OBJECTCONTEXT) could have been omitted.

7.8 Conclusions

We believe to have designed a language which enables the expression of the concepts and ideas driving the statistical framework in an intuitive and concise manner. Moreover, the designed language seamlessly integrates with LARVA, which fits in perfectly with the spirit of a tight integration between the statistical and runtime verification framework.

In essence, we have reduced the description of non-trivial statistics collection to a few lines of code. Moreover, the intuitive use of statistical events as well as the abstraction of context (from statistical events) greatly aids in the intuitive integration of statistics, runtime verification and the notion of context. The introduction of statistics and interval definitions also aids in reusability and conciseness within our language. Perhaps the least straightforward aspect of our language is the description of overlapping intervals, due to the copious amount of context definition required. While we recognise this issue, it is unfortunately necessary due to the underlying LARVA mechanism for handling context.

8. Implementation Details

With the statistical logic and associated framework thoroughly motivated, an operational semantics for the logic defined, and a language specified for the expression of the statistical framework, the next logical step is the implementation of a tool capable of generating a physically executable artefact. Using all the knowledge presented in the previous chapters, this executable artefact is to embody the expressed (by the user) notion of statistics collection at runtime specified in the form of a LarvaStat script. The following chapter presents a Java implementation of such a tool, aptly named the LarvaStat compiler[1].

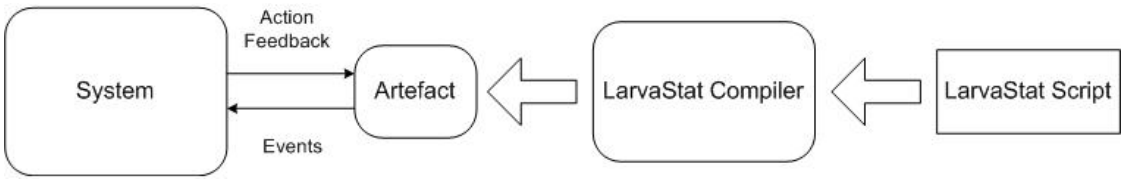


Figure 8.1: System Overview

Analogously to LARVA [12], a user which is to use the LarvaStat compiler must develop two artifacts; the underlying system and the LarvaStat script. Whereas the script is necessary for the expression of statistics collected at runtime within our framework, the underlying system is required as the basis from which the statistical framework collects statistics over its runtime execution. We identify two major issues for the development of the LarvaStat compiler

- Translation of the LarvaStat script into an executable form of code.
- Instrumentation of the translated executable code with the underlying system.

In other words, not only must the compiler generate semantically equivalent code (to the original script) which is executable, but also code which is instrumentable with the underlying system under consideration. Note that henceforth we shall be opting for automatic instrumentation, since manual instrumentation is error prone and possibly tedious. Various approaches are available for such such issues, a very viable option being that of using aspect oriented techniques. The discussion regarding such issues, as well as the overall construction of the compiler

[1]See Appendix A for the LarvaStat compiler usage instructions

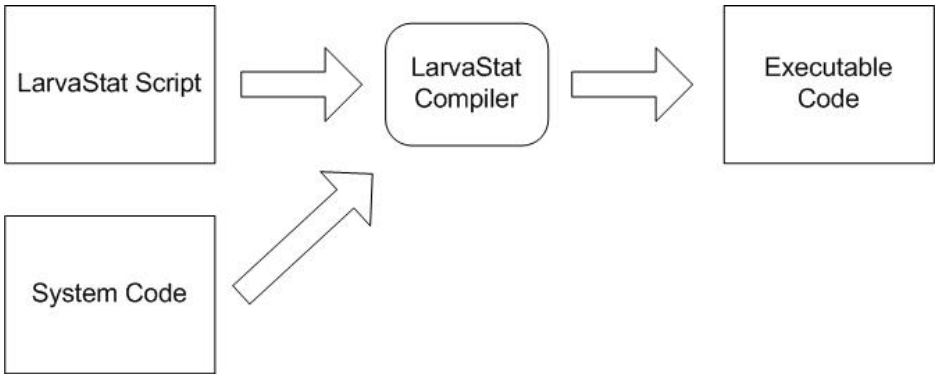
is presented in the following section.

Apart from discussing the compiler design, this chapter also motivates the mechanisms developed for the execution of the logic defined within the statistical framework. Hence, whereas the techniques used for statistical construct simulation using DATEs has already been discussed (see chapter 6), other issues such as the execution of the statistical event through implicit context declaration as well as the handling of context for overlapping intervals need still be defined.

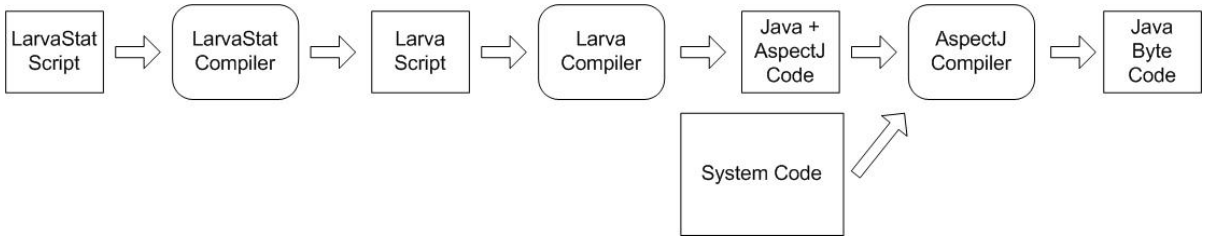
The chapter is organised as follows; section 8.1 motivates in detail the approach taken, as well as the resulting implementation overview for the LarvaStat compiler. Consequently, a discussion regarding the implementation of each of the core statistical framework notions within the compiler output is presented. Section 8.2 discusses the implementation for the statistical object, section 8.3 discusses the issues regarding the implementation of the statistical even, whereas sections 8.4 and 8.5 discuss the implementation of the point and interval statistic respectively.

8.1 The LarvaStat Compiler

We identify two approaches toward the implementation of the LarvaStat compiler, the first of which is the following



The choice of such an architecture would involve the implementation of a compiler which converts both the statistical and runtime verification frameworks into executable form. Moreover, the issue of instrumentation must also be handled by the code generated by the compiler. Hence, such an architecture would involve a considerable amount of effort, which leads us to the second proposed approach towards the implementation of the LarvaStat compiler



Using the available LARVA compiler is very advantageous. Firstly, it relieves the burden of instrumentation from the LarvaStat compiler, since the LARVA compiler produces code which (eventually — after compilation by the AspectJ compiler) is both executable and instrumentable. Secondly, this choice of architecture implies that the LarvaStat compiler can focus on statistical constructs, leaving the conversion of runtime verification properties to the LARVA compiler. Moreover, although the underlying system must still be kept in mind for the development of reasonable statistics, the LarvaStat compiler does not require any direct contact with the system. Couple all these reasons with the fact that the theory converting the statistical constructs into DATEs (which a LARVA script specifies) has already been developed makes the above architecture ideal.

Hence, the compilation of a LarvaStat script to an executable form involves multiple steps. The input script is first converted into a LARVA script (which is as far as the LarvaScript compiler goes). The semantically equivalent LARVA script is henceforth converted by the LARVA compiler into AspectJ and Java code. Given this intermediate code, as well as the underlying system code, the AspectJ compiler integrates and compiles both into a set of byte code files directly executable on the Java Virtual Machine. Thus the process of compilation terminates upon rendering the original LarvaStat script into executable and instrumented code.

Note that the design choice of using LARVA henceforth results in two major implications. The first implication is that the underlying system must be written in the Java language (since LARVA produces Java and AspectJ code, to be fed to the AspectJ compiler along with the system code). Also, the second implication is that the understanding of the LarvaStat compiler depends on knowledge of LARVA, both the defined Language as well as the inner workings of the LARVA compiler.

8.1.1 LarvaStat Compiler Design

The above choice of architecture reduces the LarvaStat compiler into an embedded language compiler, translating the additional statistical functionality present in LarvaStat back into LARVA. This translation is executed primarily according to the previously motivated conversion techniques discussed in chapter 6. The following diagram entails an overview of the LarvaStat compiler's constituent components.

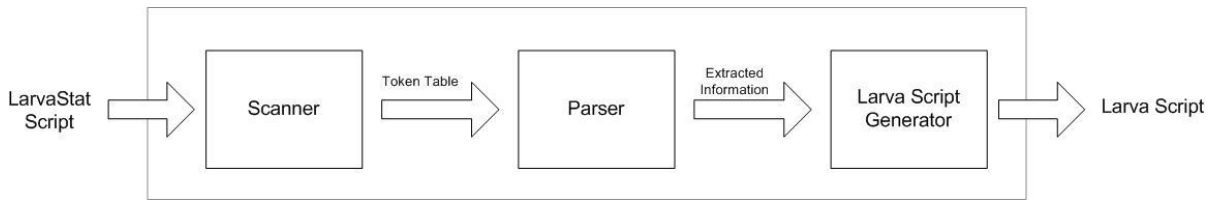


Figure 8.2: Compiler Structure Overview

Whereby the compiler follows the structure of a traditional compiler, having a scanner,

a parser and a component responsible for code generation (in our case the output code is a LARVA script). Hence, the LarvaStat script input is first tokenized during the scanning phase. Consequently, the resulting list of tokens are checked for adherence to the LarvaStat script syntax, and if so the list of tokens are organised into 'packages', with each package representing some required script information (such as constructs). Finally, the required LARVA script is generated by the LARVA script generator using the organised list of tokens passed on by the parser.

The Scanner

The scanner implemented within the LarvaStat compiler is a typical scanner implementation in the sense that it performs a lexical analysis on the input script. Hence, by looking at the script at the character level, the scanner converts a sequence of characters into a sequence of tokens. This process of recognising tokens is essentially governed by what the LarvaStat scripting language defines as reserved words and symbols. In fact, a distinct token type represents each reserved entity within the script, as well as an additional token representing general unparsed text (motivated below). Moreover, the scanner stores an additional location (within the script) associated with each token marking the location where the token was met within the script. This location is mostly used for error reporting purposes, both during the lexical and syntactic analysis phases, as will be discussed below. Consequently, each token created within the scanner is structured as follows

- The token type.
- The token contents.
- The location in the script where the token was extracted from.

Note that during lexical analysis, any comment is discarded from the script. With the process of lexical analysis complete, we end up with a sequence of tokens (assuming no error was reported). This token sequence is then passed on to the next phase, the syntax analyser (or parser) whose main task is to ensure that the token sequence adheres to the defined LarvaStat script BNF.

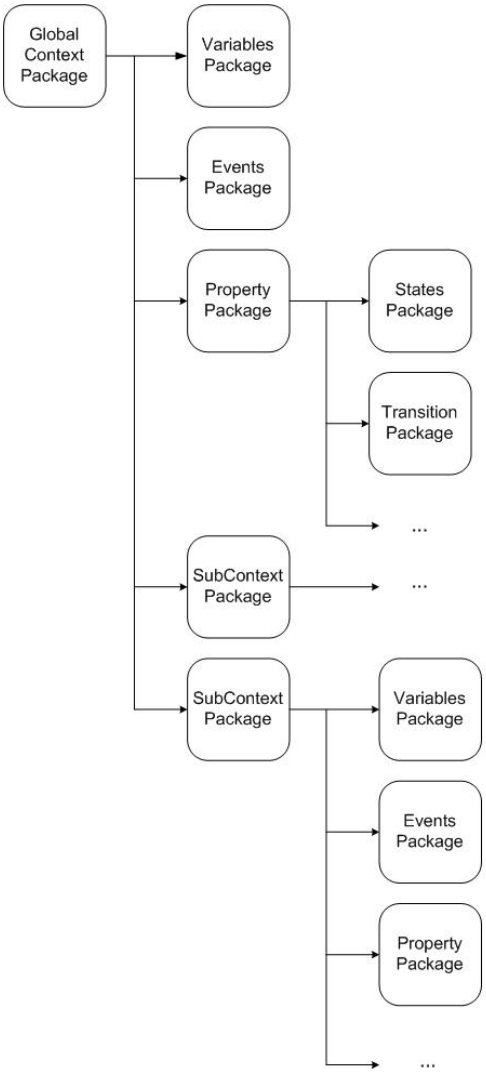
The Parser

The implemented parser is essentially a recursive descent parser implemented through mutually recursive functions. Moreover, each recursive function approximately corresponds to a non-terminal symbol within the defined LarvaStat BNF. Hence, given non-terminals such as the point statistic, interval statistic and interval declaration, the parser implements functions representing production rules (as specified by the BNF) which recursively break down such non terminals ultimately into terminal symbols.

A crucial issue regarding the execution of the syntactic analysis phase is how the parser handles the embedded language compiler's nature. Whereas the compiler is only directly interested in selective portions of the script related to statistics collection, chapter 5 motivated the

need for both frameworks being mutually dependent. This implies that although the LarvaStat compiler is not directly responsible for parsing sections related to runtime verification, in truth the compiler requires some limited parsing of such sections in order to collect information related to statistical constructs, as well as to alter certain constructs which depend on statistics collection (as motivated later). The result is a parser selectively alternating behaviour between a full parser (when parsing statistical constructs) and a shallow parser collecting just enough information as is required (when parsing runtime verification constructs).

The parser's responsibility, apart from syntactically recognising the input script, is also to extract the information required by the larva script generator. In other words, if a token sequence is for example recognised to represent a point statistic declaration, then the information implied by this token sequence is extracted, 'packaged', and ultimately passed as part of the parser output. These information packages are in fact implemented as objects internal to the compiler. Since the core constructs are recursively defined (such as the definition of context), the parser outputs is a tree-like information package structure, with the uppermost package denoting the global context as exemplified below



Hence using such an information package structure which is built up during parsing, all the parser is required to output is the global context, since the global context in fact contains all the information defined within the script. In truth the parser also outputs an additional two tokens representing information not defined within the global context. These tokens represent the imports and method declaration sections. Note that using such a package structure faithful to the original script's conceptual structure implies that the statistical ordering defined within the script remains intact.

Whereas statistical constructs are fully parsed, most other sections (such as the declared variable sections or method declarations) are not, and are hence represented as a general text token and directly passed to the larva script generator. In truth, the only other constructs which are parsed to a degree are those which possibly affect or are affected by the statistical constructs, which in our case are the property declarations. Although not fully parsed, each property may listen to statistical events, in which case certain alterations to the property definition may be

required, as will be motivated below.

The LARVA Script Generator

The final phase within the LarvaStat compilation process entails the generation of LARVA code. The LARVA script generator essentially uses the information package tree-like structure output by the parser in order to generate a script consisting solely of DATE constructs expressed using LARVA. Since these information packages also contain information regarding statistical constructs (hence not inherently part of LARVA), such constructs are converted to DATE constructs using the techniques discussed in chapter 6.

The LARVA script generator component traverses each package output by the parser. In most cases, if the particular package denotes a construct defined in LARVA (such as the declaration of variables), the script generator produces the same code unaltered. However, in the case that the construct denotes a property, the script generator checks if any defined transition listens on a statistical event. If so, each such transition definition is slightly altered to implicitly (without the user's knowing) handle the statistical event context as defined below. However, if no transition listens on a statistical event the property also remains unaltered. If the information package denotes a statistical construct the script generator produces a set of additional constructs, these being (i) required channels and events for the simulation of the statistical events (ii) a statistical object (in the appropriate context), and (iii) a semantically equivalent DATE construct (in the appropriate context). The generation of all three constructs will be discussed in further detail below. Note that any construct created by the script generator which is not visible to the user is declared with a $ symbol at the start of the construct name.

Due to the recursive nature of the LARVA BNF, as well as the tree-like structure output by the parser, the LARVA script generator is defined through a set of mutually recursive functions. Through this recursive definition, the LARVA script generator performs a depth-first traversal on the parser output, processing each node encountered as specified above, and generating LARVA code accordingly. Note that considerable care was taken in the design of the script generator for the production of user friendly, readable code. This is essentially achieved through the implementation of a mechanism which automatically calculates the appropriate indentation, as well as through copious automatically generated comments. The main motivation for user friendly code is for debugging purposes, since a user knowledgable in LARVA can manually read through the generated LARVA script should the requirement arise.

Error Reporting

The LarvaStat compiler can encounter errors during the compilation process. These errors take two forms, these being syntax and semantic errors. The compiler is built with the issue of error reporting kept in mind. Hence, an intuitive mechanism for the reporting of errors is implemented within the compiler. Moreover, a considerable list of errors (both syntactic as well as semantic) have been identified. Although the whole list of error codes will not be discussed (see appendix B for a full listing), we broadly organise errors into four groups; syntactic errors

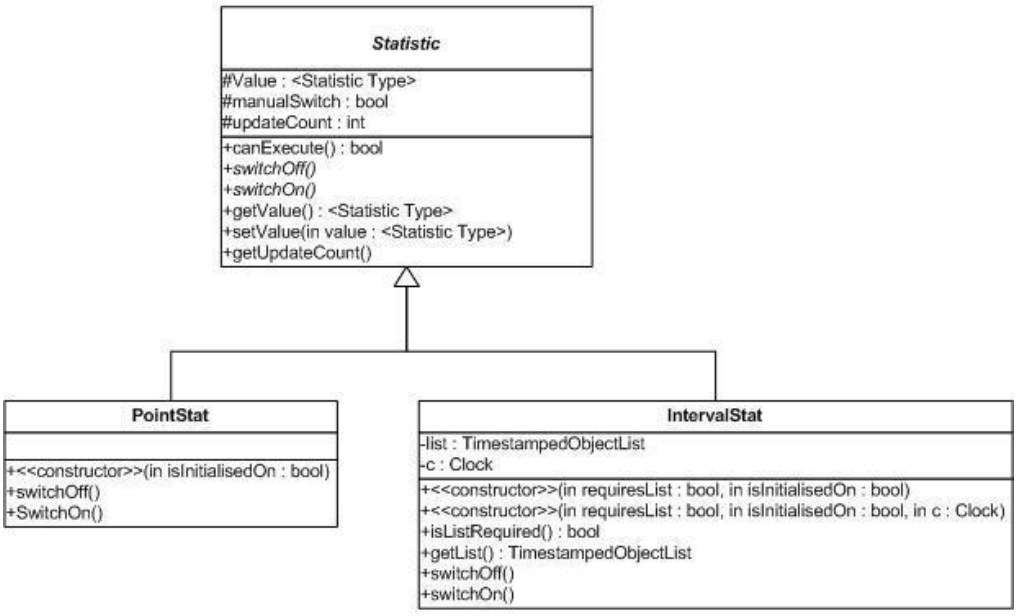
regarding missing constructs, syntactic errors regarding missing reserved words, other miscellaneous errors and finally semantic errors.

Whenever an error is encountered during compilation, an error code is generated specifying the error encountered. In most error cases (especially those of a syntactic nature), the location within the script (extracted from the erroneous token) is also passed to the error reporting mechanism, which given the information generates an error message to the user. Note that an error can be encountered during ay of the three compilation phases.

This error reporting mechanism is implemented through the implementation of the LarvaStatCompilerException exception class, whose overloaded constructor accepts an error code and possibly a script location and in return generates an appropriate error message and triggers an exception.

8.2 Statistic Object Implementation

The implementation of the statistic object is defined by previously motivated class structure described below

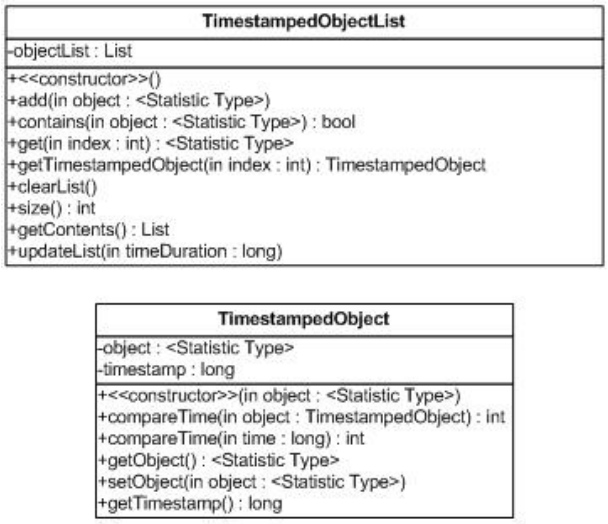


Hence, three classes are implemented in all; the Statistic abstract class, as well as the PointStat and IntervalStat classes. All declared methods are self explanatory, whereby apart from the getter and setter methods, method canExecute() returns the manual switch state, and the switchOn() and switchOff() methods define the appropriate logic required for switching on or switching off statistic evaluation. Hence, whereby the point statistic simply alters the manual switch state, the interval statistic also pauses or resumes the associated Clock object (should any be defined), as specified below

```
public void switchOff(){
    manualSwitch = false;
    if (c != null)
       c.pause();
}
```

```
public void switchOn(){
    manualSwitch = true;
    if (c != null)
       c.resume();
}
```

Note that given the UML structure above, the IntervalStat class requires the implementation of the TimestampedObjectList class. Moreover, the TimestampedObjectList class requires the definition of the TimestampedObject class, which is responsible for tagging an object with a timestamp marking the time of its creation. Both classes are described below



Both classes are trivial in nature, with the TimestampedObjectList class defining most of its logic through actions on the private List object. However, a method of interest is the updateList method, responsible for updating the internal list as defined below

```
public void updateList(long timeDuration) {
    for(int i = 0; i < objectList.size(); i++) {
        if ((System.currentTimeMillis() - (objectList.get(i).getTimestamp()))
                                                        > (timeDuration*1000)) {
            objectList.remove(i);
            i--;
        }
    }
}
```

whereby each element in the internal list whose timestamp is older than the duration of time passed as a parameter is discarded.

Of particular note is the use of generics for the solution of the issue of altering certain variable types in accordance with the declared statistic type. In other words, since the statistic type can vary, we require a mechanism whereby the the LarvaStat compiler sets the variable type for that variable responsible for storing the statistic valuation (and associated methods). An obvious solution is achieved by defining this variable to type Object and extensively using typecasting throughout the script, such as every time the statistic valuation's getter and setter methods are called. However, we present a better solution, whereby all specified classes are defined as generic classes as exemplified below

```
public class PointStat<T> extends Statistic<T>{
...
}

public class IntervalStat<T> extends Statistic<T>{
```

```
...
}
```

Generics as defined in Java move type checking at runtime implemented through typecasting to checking at compile time. This allows the LarvaStat compiler to generate text specifying the type that particular statistic declaration requires, thus removing the requirement of typecasting since all variable types are known at compile time. The following are a few statistic object instantiations as specified using generics

```
PointStat<Integer> totalBytesDownloadedPerUser = new PointStat<Integer>(true);

IntervalStat<Double> TimeElapsedCount = new IntervalStat<Integer>(false,true);
```

The type of both statistics are defined through a tag at the end of the class name. Such declarations need not be declared by the user in the LarvaStat script, but are generated automatically by the compiler in the output LARVA script for the representation of the notion of the statistic object as motivated previously. Note that the use of generics results in two implications, the first being that a J2SE 5.0 compliant Java Virtual Machine is required for the execution of the resulting code, and also that only reference type objects are allowed for the definition of statistic types, since generics do not permit the use of primitive types.

8.3 Statistical Event Implementation

We identify two issues of interest for the implementation of the statistical event, these being the generation of the required constructs for the simulation of the statistical event, and the implementation of the internal mechanism abstracting the notion of context as motivated previously.

The generation of the required constructs follows the technique motivated in section 6.1. Hence, a fresh channel, event construct pair is defined for each declared statistic. Note that whereas the event declaration represents the statistical event, hence available to the user and defined under the name of ⟨Statistic Name⟩ + "_Event" (as previously stated), the channel responsible for statistic valuation transfer is hidden from the user, and will thus be defined with an initial '$' symbol. Also, since we require that the value triggering the statistical event is available to the user, an additional variable of the same type as the statistic is also declared, whose valuation is always guaranteed to be the statistical valuation which triggered the statistical event. Given an example point statistic totalBytesDownloaded storing the byte count downloaded, the following constructs are defined for the simulation of the statistical event

```
...
VARIABLES{
    Channel $totalBytesDownloaded_Channel  = new Channel();
    Integer totalBytesDownloaded_Value;
    ...
}
EVENTS{
```

```
    totalBytesDownloaded_Event(ContextObject $totalBytesDownloaded_Obj)
       = {$totalBytesDownloaded_Channel.receive($totalBytesDownloaded_Obj)}
    ...
}
...
```

Note that the ContextObject object is defined as part of the mechanism for handling context, and will be discussed below. Also note how using the $ symbol the only constructs visible to the user are the value passed on the channel and the event itself simulating the statistical event. The channel and context object are hidden from the user. Moreover, if the statistic is an event or duration interval statistic, then an additional set of constructs is defined for the simulation of the closing interval event (motivated in section 7.4). Given event interval statistic BytesDownloadedPerConnection, the resulting constructs for the simulation of both events are

```
...
VARIABLES{
    Channel $BytesDownloadedPerConnection_Channel  = new Channel();
    Integer BytesDownloadedPerConnection_Value;
    ...
    Channel $BytesDownloadedPerConnection_ChannelComplete  = new Channel();
    Integer BytesDownloadedPerConnection_FinalValue;
    ...
}
EVENTS{
    BytesDownloadedPerConnection_Event(
               ContextObject $BytesDownloadedPerConnection_Obj)
               = {$BytesDownloadedPerConnection_Channel.receive(
                                          $totalBytesDownloaded_Obj)}
    ...
    BytesDownloadedPerConnection_EventComplete(
               ContextObject $BytesDownloadedPerConnection_Obj)
               = {$BytesDownloadedPerConnection_ChannelComplete.receive(
                                          $totalBytesDownloaded_Obj)}
}
...
```

Recall that section 5.11 motivated the need for a construct to be able to listen to the implicit statistical event implied by any statistic declaration regardless of context, as long as the statistic and construct are defined within nested contexts. Also recall that listening to a statistical event requires no explicit declaration of context by the user, which significantly simplifies matters from the user's point of view. Nevertheless, any event declared within LARVA requires the handling of context, which implies that context for statistical events must be internally handled by the resulting LARVA script. This is achieved through the development of a mechanism which automatically deduces the context for each construct listening to the statistical event.

Since we require that the statistical event is visible to all contexts regardless of where the statistic is declared, this implies that all created constructs for the simulation of the statistical events need to be visible to all contexts. In order to achieve this all constructs are declared

within the global section, since constructs defined within the this section are visible to all declared subcontexts. Moreover, such a design choice relieves the event declaration simulating the statistical event from explicit context declaration (since the GLOBAL context is inherently not bound by any particular context). However, statistical events are not defined as broadcast events triggering all constructs listening on the event, but rather as events triggering constructs within the same nested context structure. This leads us to an alternate handling of context through a mechanism specified below.

At the core of this mechanism lies the ContextObject class, whose instances are sent over the channels used to simulate statistical events. The ContextObject class essentially encapsulates the statistical valuation with the context the statistic is declared within, as defined below

```
public class ContextObject {
    private ArrayList ContextList;
    private Object obj;
    ...
}
```

Whereby each time a statistical event triggers, both the statistical valuation and the associated contexts are passed over the appropriate channel. Hence, simulating the statistical event through an implicit context is reduced to (i) developing a mechanism which generates appropriate context objects, and (ii) using the context information to determine which constructs are applicable. Generating appropriate context objects is trivial, whereby the following code is used each time a statistical event is triggered

```
Object[] contexts = {<Statistic Contexts>};
$<Statistic Name>_Channel.send( new ContextObject(contexts,
                                        <Statistic Name>.getValue());
...
```

Note that the statistic contexts tag refers to the set of context objects characterising the subcontext which the defined statistic is situated in. Hence, given the following statistic declaration

```
GLOBAL{
    ...
    FOREACH (Session s) {
    ...
        FOREACH (Connection c) {
        ...
        <Statistic Declaration>
        ...
        }
    }
}
```

the contexts array is defined as contexts = {s,c}. Note that the construction of such information is available through the recursive context traversal of the information package structure

parser output.

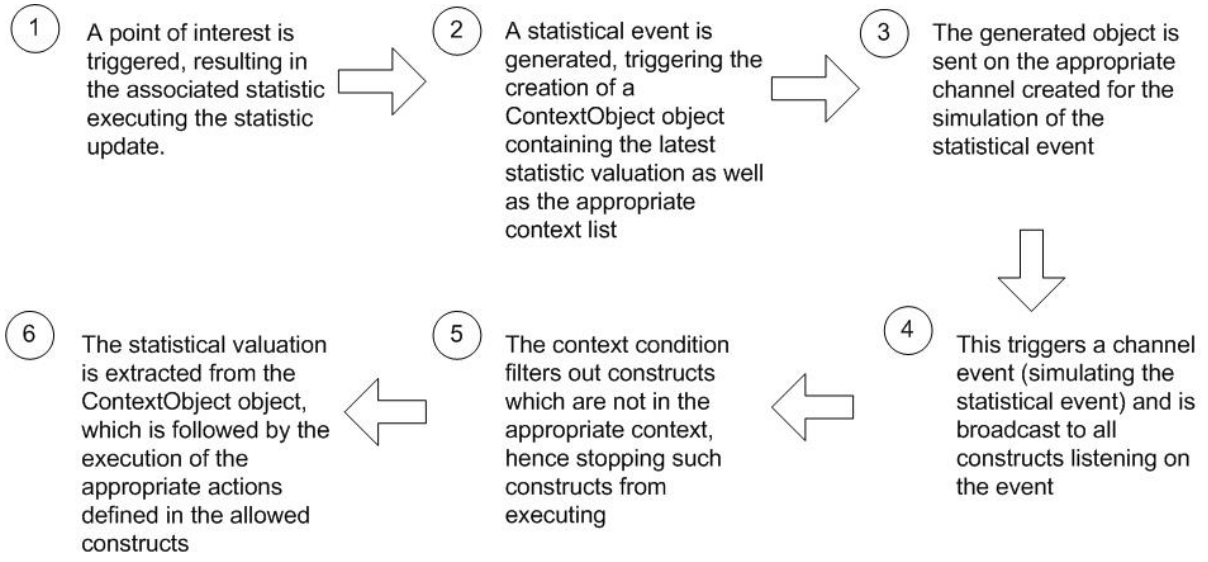
Since all events simulating the statistical events are defined in the GLOBAL section, context determination can not be specified through the event declaration. An alternate approach taken is through the definition of a condition for each transition triggering on each statistical event. This condition ensures that only constructs in the appropriate context are triggered by the event, and is constructed using the statistical event context rules defined in section 7.4. Hence, the resulting condition uses enough context information from the context object as is available, or required. Given a context array (sent over the context object) contexts = { c_1, c_2, c_3 ... c_n } — where the declared statistic responsible for the statistical event is declared in the n^{th} subcontext — as well as a transition listening on this statistical event defined in the i^{th} context

- if (i < n) condition is
  ((c_1 == contexts[1]) && (c_2 == contexts[2]) && ... && (c_i == contexts[i]))
- if (i ≥ n) condition is
  ((c_1 == contexts[1]) && (c_2 == contexts[2]) && ... && (c_n == contexts[n]))

Hence, a statistical event acts as a broadcast event for constructs declared in contexts below the statistic declaration context, since in such cases the context mechanism can only use as much context information as was passed by the context object, and cannot distinguish between lower contexts. Given the following condition construction, any transition listening on a statistical event is generated as follows

```
[ <Statistic Name>_Event \ ((($c_1$ == contexts[1]) && ($c_2$ == contexts[2])
                           && ... && ($c_i$ == contexts[i])) &&
                           <transition condition>) \
                           <Statistic Name>_Value =
                               (<Statistic Type>)$<Statistic Name>_Obj.getObj();
                           <transition action> ]
```

Note that <transition condition> and <transition action> refer to the condition and action defined for the same transition within the LarvaStat script. Given a transition listening on a statistical event, the <Statistic Name>_Value variable (declared previously) is set to the statistic valuation extracted from the context object prior to the transition action execution. This ensures that any transition action has access to the statistical valuation which triggered the statistical event. The following diagram summarises the process of context resolution as defined above



Since transitions are the only constructs defined within LARVA which trigger on events, the compiler checks each transition generated in the LARVA script if it listens on a statistical event, and if so applies the context resolution mechanism accordingly. Moreover, since the defined statistical constructs are also converted to DATEs defined through transitions, this process is also applied to DATEs representing statistics (since statistics themselves can listen to statistical events). In truth, it is this whole process of abstracting context from the statistical event which lead to the requirement of the LarvaStat compiler to parse property declarations (as defined previously). Nevertheless, most importantly a non-trivial issue of context resolution has been automated and hence completely abstracted from explicit implementation, thus relieving the user from certain complexity. Had this not been the case, the user would have had to implement a similar mechanism for each statistical event required.

8.4 Point Statistic Implementation

Implementation of the point statistic conversion involves two steps: (i) create the statistic object, and (ii) convert the point statistic into a DATE construct. Note that by the time a point statistic requires conversion, the required constructs for the simulation of the statistic event associated with this point statistic would have already been created in the global context. Conversion of the point statistic essentially involves the implementation of previously motivated techniques. Hence note that both steps have been previously covered, with the creation of the statistic object discussed in section 8.2, whereas the conversion technique used for the conversion of the point statistic discussed in section 6.2. The following example illustrates the complete conversion of point statistic totalLoginCount

```
GLOBAL{
    ...
    FOREACH (Session s)
    {
```

```
        ...
        POINTSTAT totalLoginCount : Integer
        {
            INIT{ totalLoginCount.setValue(new Integer(0));  }

            EVENTS{        userLogin() }

            CONDITION{ result.equals("230 logged in") }

            UPDATE{ totalLoginCount.setValue(totalLoginCount.getValue() + 1); }
        }
        ...
   }
   ...
}
```

where point statistic totalLoginCount keeps count of the number of successful login attempts for each created session. Note that variable result is extracted from the system by the userLogin() event declaration. The above point statistic declaration is a typical construct declaration within the LarvaStat script. However, since LARVA defines no such notion of statistics, the above declaration is converted to the following semantically equivalent LARVA code.

```
GLOBAL{
    VARIABLES {
        Channel $totalLoginCount_Channel  = new Channel();
        Integer totalLoginCount_Value;
        ...
    }
    EVENTS {
        totalLoginCount_Event(ContextObject $totalLoginCount_Obj) =
                    {$totalLoginCount_Channel.receive($totalLoginCount_Obj)}
        ...
    }
    ...
    FOREACH (Session s) {
        VARIABLES {
            PointStat<Integer> totalLoginCount = new PointStat<Integer>(true);
            ...
        }
        ...
        %% Property representing POINTSTAT [Name: totalLoginCount,
                                             Statistic Type: Integer]
        PROPERTY totalLoginCount{
            STATES{
                BAD { } NORMAL { }
                STARTING { start { if (totalLoginCount.getValue() == null){
                            totalLoginCount.setValue(new Integer(0));  ; } } }
            }
            TRANSITIONS{
                 start -> start
```

```
                [
                userLogin()  \
                ((result.equals("230 logged in") ) &&
                                        (totalLoginCount.canExecute())) \
                totalLoginCount.setValue(totalLoginCount.getValue() + 1);
                Object[] contexts = {s};
                $totalLoginCount_Channel.send( new ContextObject(
                                    contexts, totalLoginCount.getValue()));
                ]
            }
        }
        ...
    }
}
```

An immediate observation is the substantial amount of logic implied by a single point statistic declaration. Therefore, since the point statistic is one of the simpler statistical constructs it is envisaged that the logic implied by other constructs is even more substantial, as will be discussed below. The above point statistic conversion is essentially the culmination of previously motivated techniques. The required channels, events and variables for the simulation of the statistical event are created in the global context. The statistic object representing the point statistic are created in the same context where the statistic is defined, and the point statistic construct is converted into a one-state, one-transition DATE (as motivated in section 6.2. This transition is responsible for the execution of the triggering of the point of interest, as well as the execution of the statistic update. Note that code for statistic initialisation is associated with the initial state (and is executed only upon initialisation). Also, an additional totalLoginCount.canExecute() condition is added to the transition condition in order to check if the statistic has been switched off through an action on the statistic object. After the statistic update execution, a statistical event is generated by creating a ContextObject object (containing the appropriate context list and statistical valuation), and sending this object on the appropriate channel. Any transition listening on the associated event is hence triggered, thus simulating the statistical event semantics.

8.5 Interval Statistic Implementation

The approach for the implementation of the conversion of interval statistics primarily depends on the issue of overlapping intervals. Hence, two distinct techniques are implemented, one for the conversion of interval statistics with non-overlapping contexts, and another for interval statistics admitting overlapping contexts.

The implementation for interval statistics with non overlapping intervals is analogous to the point statistic conversion discussed above. Hence, its implementation involves the creation of the statistic object and the conversion of the interval statistic to a DATE construct using the appropriate conversion technique. Similarly to the point statistic, all constructs required for statistical event simulation associated with the statistic are created within the global context beforehand. Note that since the interval statistic inherently implies richer semantics than the

point statistic, the resulting LARVA code is even more substantial than the point statistic equivalent. Also, since the time interval statistic inherently does not admit overlapping intervals, all time interval statistics are converted using this technique.

In the case of time interval statistics, we require the definition of the polling duration δ (defined in section 5.4.4). Theoretically, the smaller δ is the more frequently up to date will the time interval statistic be. However, although there is no limit to the size of δ, in truth making δ too small creates a substantial overhead on the system. The reason for this is that the smaller δ is, the more frequently is the polling update action executed. This situation is compounded by the fact that clocks are implemented in LARVA using threads, and Java thread scheduling becomes rather unreliable for scheduling at very short time notice. Hence, keeping such issues in mind as well as through practical experience, we have chosen $\delta = 0.5$s. We believe this choice reaches a balance between overhead and an acceptable updating rate. Still, the user is encouraged to design statistic updates and polling actions as efficient as possible. Nevertheless, the choice of δ is borne out of experience rather than theory, and can be easily altered in the future should the requirement arise.

Although three distinct interval statistic types are defined, all three are analogous in their implementation for their conversion (assuming non-overlapping intervals), whereby the only real difference is the resulting DATE structure defined by the conversion techniques motivated in chapter 6. We shall hence explain the first technique through the example conversion of time interval statistic bytesDownloadedLast30Minutes, which in turn listens to a statistical event defined for event interval statistic BytesDownloadedCountPerConn (discussed for the next technique) — hence defining a two-layer statistic declaration.

```
GLOBAL{
    ...
    FOREACH (Session s)
    {
        ...
        INTERVALSTAT bytesDownloadedLast30Minutes : Integer
        {
            INIT{bytesDownloadedLast30Minutes.setValue(new Integer(0));}

            EVENTS{BytesDownloadedCountPerConn_EventComplete}

            LIST{true}

            INTERVAL{
                TIME[1800 \  ]
            }

            UPDATE[userDownloadsLast30Minutes.getList().add(
                   new Integer(BytesDownloadedCountPerConn_FinalValue)); ]
                  { Integer count;
                    for(Integer i : bytesDownloadedLast30Minutes.getList()
                                                          .getContents())
```

```
                    {count += i; }
                bytesDownloadedLast30Minutes.setValue(count);
            }
        }
        ...
    }
    ...
}
```

Interval statistic BytesDownloadedCountPerConn evaluates the number of bytes downloaded for each created connection. Each time a connection is terminated marks the closing of the interval statistic, thus triggering a statistical event marking the interval closing. Statistic bytesDownloadedLast30Minutes listens on this statistical event, adding each value sent on this statistical event (marking the number of bytes downloaded for that particular connection) to the interval statistic's list as defined by the exclusive update. Implicitly, this list is updated during polling, thus removing any statistical valuations added to the list older than 30 minutes (1800 seconds). The statistical update involves adding all the statistical valuations received within the last 30 minute period, thus resulting in the total number of bytes downloaded within this period. Statistic BytesDownloadedCountPerConn is converted to the following equivalent LARVA code

```
GLOBAL{
    VARIABLES {
         Channel $bytesDownloadedLast30Minutes_Channel  = new Channel();
         Integer bytesDownloadedLast30Minutes_Value;
         ...
    }
    EVENTS{
        bytesDownloadedLast30Minutes_Event(ContextObject
                        $bytesDownloadedLast30Minutes_Obj) =
                        {$bytesDownloadedLast30Minutes_Channel.receive(
                                    $bytesDownloadedLast30Minutes_Obj)}
        ...
    }
    ...
    FOREACH (Session s)
    {
        VARIABLES {
            Clock $bytesDownloadedLast30Minutes_Clock = new Clock();
            IntervalStat<Integer> bytesDownloadedLast30Minutes =
                                    new IntervalStat<Integer>(true,true,
                                       $bytesDownloadedLast30Minutes_Clock);
            ...
        }
        EVENTS {
            $bytesDownloadedLast30Minutes_Update()
                            = {$bytesDownloadedLast30Minutes_Clock @ 0.5}
            ...
        }
```

```
        ...
        %% Property representing INTERVALSTAT [Name: bytesDownloadedLast30Minutes,
                                    Statistic Type: Integer, Interval Type:  Time]
        PROPERTY bytesDownloadedLast30Minutes{
            STATES{
                BAD { } NORMAL { }
                STARTING {
                    start {if (bytesDownloadedLast30Minutes.getValue() == null){
                    bytesDownloadedLast30Minutes.setValue(new Integer(0));} } }
            }
            TRANSITIONS{
                start -> start[
                    $bytesDownloadedLast30Minutes_Update() \
                    (bytesDownloadedLast30Minutes.canExecute()) \
                    bytesDownloadedLast30Minutes.getList().updateList((long)1800.0);
                    Integer count;
                    for(Integer i : bytesDownloadedLast30Minutes
                                                .getList().getContents())
                    {count += i; }
                    bytesDownloadedLast30Minutes.setValue(count);
                    $bytesDownloadedLast30Minutes_Clock.reset();
                ]
                start -> start[
                    BytesDownloadedCountPerConn_EventComplete \
                    ((((s == (Session)$BytesDownloadedCountPerConn_Obj.
                    getContextList().get(0)))) &&
                                 (bytesDownloadedLast30Minutes.canExecute())) \
                    BytesDownloadedCountPerConn_FinalValue =
                                 (Integer)$BytesDownloadedCountPerConn_Obj.getObj();
                    bytesDownloadedLast30Minutes.getList().updateList((long)1800.0);
                    userDownloadsLast30Minutes.getList().add(
                                new Integer(BytesDownloadedCountPerConn_FinalValue));
                    Integer count;
                    for(Integer i : bytesDownloadedLast30Minutes.getList()
                                                                .getContents())
                    {count += i; }
                    bytesDownloadedLast30Minutes.setValue(count);
                    Object[] contexts = {s};
                    $bytesDownloadedLast30Minutes_Channel.send(
                                new ContextObject(contexts,
                                bytesDownloadedLast30Minutes.getValue()));
                ]
            }
        }
        ...
    }
    ...
}
```

The resulting LARVA script is indeed substantial, although analogous to the previous point

statistic conversion. Using the time interval statistic conversion technique, the result is a one-state, two-transition DATE. Whereas one transition is responsible for implementing the polling mechanism, the other is responsible for implementing the point of interest and statistic update. Note that the statistic makes use of an additional timer (analogously to the duration interval statistic). This internal timer is created within the same context as the statistic declaration. Moreover, an associated timer event is created which triggers every 0.5s (the chosen value for the polling duration). This event is used to drive the polling mechanism, whereby the list is updated and the statistic update executed every 0.5s. The other transition listens to a statistical event, which results in the handling of a statistical event as discussed previously. Hence, a context condition is inserted, as well as the value from the ContextObject object extracted first in the transition action. Consequently, the list is updated and statistic update executed, with the action finally generating a statistical event to mark the creation of a new statistic valuation.

The following example illustrates the second technique for the conversion of interval statistics which admit overlapping intervals. Instead of defining the full conversion, this example focuses on introducing the additional logic required for the execution of overlapping intervals. Hence, issues such as the constructs required for the conversion of the statistical event shall be omitted.

```
GLOBAL
{
    ...
    FOREACH (Session s)
    {
        ...
        INTERVALSTAT BytesDownloadedCountPerConn : Integer
        {
            INIT{BytesDownloadedCountPerConn.setValue(new Integer(0));}

            EVENTS{sendInfo}

            CONDITION{ out == conn.bOut }

            LIST{false}

            INTERVAL [CONTEXT (Connection conn)]{
                OPEN[downloadStarting \ \ ] WHERE {c == conn}
                CLOSE[downloadComplete \ \ ] WHERE {c == conn}
            }   OBJECTCONTEXT { s = conn.getSession(); }

            UPDATE{ BytesDownloadedCountPerConn.setValue(
                          BytesDownloadedCountPerConn.getValue()
                                                + conn.bufferSize); }
        }
        ...
    }
}
```

Statistic BytesDownloadedCountPerConn is an event interval statistic admitting overlapping intervals, where each connection created (within a session) is associated with a fresh instance of

the statistic. This logic is achieved through the definition of context which is associated with the interval declaration (seen above). Note that variable c is extracted from the system by events downloadStarting() and downloadComplete(), and refers to the connection object which triggered the download event. This variable is used to distinguish between multiple overlapping interval instances within the WHERE clause. Also note that variable out is extracted by the sendInfo() event, and denotes the output connection stream on which the byte was sent. Whereas the context can be deduced for the opening and closing of the event, each time a byte is sent (thus triggering event sendInfo()) all interval statistic instances are triggered since no context is defined for the point of interest. Hence, we shall be using the statistic condition as a context filter, disallowing statistics from executing whose associated output connection stream (extracted from the connection) is not equal to that used for the sending of the byte during the event. The resulting mechanism for the execution of overlapping intervals is the following

```
GLOBAL
{
    ...
    FOREACH (Session s)
    {
        %% Context representing INTERVALSTAT [Name: BytesDownloadedCountPerConn,
                                    Statistic Type: Integer, Interval Type:  Event]
        FOREACH (Connection conn)
        {
            VARIABLES {
                IntervalStat<Integer> BytesDownloadedCountPerConn
                                          = new IntervalStat<Integer> (false,true);
            }
            EVENTS {
                $Connection_ObjectCreated() = {execution
                                                  Connection $Connection_newObj.new()}
                                                where {
                                                  conn = $Connection_newObj;
                                                  s = conn.getSession(); }
            }
            PROPERTY BytesDownloadedCountPerConn{
                STATES{
                    BAD { } NORMAL { withinInterval }
                    STARTING { start {...} } }
                }
                TRANSITIONS{
                    start -> withinInterval[ downloadStarting  \
                                            ((BytesDownloadedCountPerConn.canExecute())
                                              && (c == conn)) \ ]
                    withinInterval -> withinInterval[ sendInfo \
                                                  ((BytesDownloadedCountPerConn
                                                              .canExecute())
                                                  && (out == conn.bOut)) \
                                                  ...]
                    withinInterval -> start[ downloadComplete  \
                                            ((c == conn) &&
```

```
                                                    (BytesDownloadedCountPerConn
                                                                    .canExecute())) \
                                                    ...]
                }
            }
        }
    }
}
```

The result is the creation of a nested FOREACH statement bound to a connection object instance. Moreover, of central importance is the definition $Connection_ObjectCreated() event declared within this nested construct. Although not directly referred to by any transition, it is actually crucial for the creation of a new context instance upon creation of a new connection. In fact, this event declaration is required due to the LARVA inner mechanism for the creation of new contexts, since LARVA actually creates a new context upon the triggering of an event whose associated context has not already been met by the LARVA context mechanism(see dynamic triggers as discussed in [12]). However, this mechanism checks for context creation only at the context level where the event is declared, taking no notice of sub contexts listening on the event. This would imply that no context instance is ever declared for the automatically generated nested FOREACH context, which is why we require the declaration of an event in order to create such instances of the nested context. Event $Connection_ObjectCreated() actually defines the event encapsulating the connection constructor call (which is why we require interval context objects to explicitly define a constructor). As defined in LARVA, since this event is nested within contexts, we require the event to provide the whole context. Whereas the context linking the event to the connection is automatically generated, we require the OBJECTCONTEXT parameter so that the user can define the remainder of the required context. Hence, in this case the remainder of the context is specified by linking the connection with the responsible session, specified by the OBJECTCONTEXT parameter as s = conn.getSession() and added to the where clause of the same event.

Hence, the above mechanism creates a new context for each new connection object, resulting in a correct context-interval statistic instance mapping required for the solution of overlapping intervals. Nevertheless, we still require a mechanism which distinguishes between interval contexts for events of interest to the interval. As exemplified above, this is achieved through the implementation of conditions acting as context filters. Hence, using the context specified in the WHERE clause defined by the user for interval points of interest (in the above case specified as (c == conn)), such conditions distinguish whether the connection triggering the event is the same as that defined as the interval context object. Hence, the above example implements three context filters, one per transition, where all three conditions filter interval statistic instances whose associated connection is not the same as that triggering the particular transition event. An important observation is that whereby the OBJECTCONTEXT parameter requires the expression of the complete context, using the above implementation of context filters the WHERE clause need only define context so as to distinguish between the interval context object.

8.6 Conclusions

This chapter provides further testament that the development of our statistical framework is worthwhile, since even the simplest of statistical constructs require a comprehensive amount of logic for their simulation in LARVA. This implies that complex statistical logic has been reduced to a trivial definition, essentially removing the requirement of all logic which can be automatically generated. Hence, all that is left to the user is the specification of the statistic's nature, rather than how the statistic is to be executed.

The implementation presented above is essentially created for the purpose of proof of concept, whereby using the this implementation concepts and techniques discussed in the previous chapter can be expressed and executed on real life scenarios as will be presented in chapters 10 and 11. With regards to the implementation itself, whereas we consider the implementation of non-trivial issues (such as the handling of the statistical event context as well as overlapping intervals) to be satisfactory, we recognise that the current implementation produces a considerable amount of constructs. Whereas certain constructs are necessary (as specified by the construct conversion techniques defined in chapter 6), we believe that certain other constructs can be reduced through the construction of better mechanisms. This point especially holds when considering the copious amount of channels created (one or possibly two per statistic — hence resulting in a considerable amount for even a reasonable amount of statistic declarations). While this issue is not pressing since the compiler is a proof of concept implementation, it may require looking at in the future for efficiency purposes. In truth this issue is lessened by the latest LARVA implementation simulating all clocks and channels on one thread each, however it is still applicable.

Another issue worth noting is that the choice of polling duration δ is currently based on practical experience, apart from a few other issues such as Java thread scheduling accuracy. Hence, although current results suggest good performance by the polling mechanism based on the current valuation of 0.5s, we would like to study further the polling duration's nature in order to choose its valuation based on a more formal reasoning.

Part III

Analysis & Evaluation

9. Formal Semantics

With the theory and implementation complete, the following chapter offers a formal analysis of the presented statistical framework. This analysis shall be presented through the formalisation of the notion of the interval, and is followed by the definition of an operational semantics for the defined statistical logic. Hence, the chapter is structured as follows: Section 9.1 formalises the notion behind the three interval types motivated in section 5.4, and is followed by section 9.2 which presents constructions formalising the conversion of the constructs within our statistical framework to DATE constructs, thus formalising the approach taken in chapter 6.

9.1 Intervals

The three intervals identified within our statistical framework, aptly named event, duration and time intervals form an expressive backbone with which to define a subset of the event trace of interest for the particular statistic. These interval types are summarised as follows:

- **Event Interval** — An interval, whose start is marked by the combination of the triggering of an event expression and the satisfaction of a boolean condition – a point of interest (described in section 5.2) – and ends through the triggering of another point of interest marking the closing of the event.
- **Duration Interval** — An interval, whose start is again marked by a point of interest characterising the opening of the interval, and stretches for a fixed duration of time.
- **Time Interval** — A dynamic interval in constant motion, the time interval always contains the subset of the event trace starting at a time duration prior to the current time and ending at exactly the current time.

Whereas the first two interval types are semantically similar (where the latter is reducible to the former), the third interval type has a different characterisation altogether. This distinction is also reflected in the behaviour of these intervals with respect to the issue of overlapping intervals. While all these issues have already been discussed (section 5.4), it should serve as a reminder since these issues shall be reflected in the formal semantics below. Prior to the definition of the intervals we shall be formalising a set of fundamental notions which the remainder of this chapter is dependent upon.

Definition 9.1.1 *The notion of time is defined as*

$$\mathbb{T} \stackrel{def}{=} \mathbb{R}^+ \cup \{ 0 \}$$

Thus, time is represented by a non-negative real number.

Two other notions extensively used throughout the chapter are the event and the system state. Consequently, note that henceforth we shall be reusing the formalised notion of an event as expressed for DATEs (see definition 3.2.1). By the term system state (or state in short), we mean the current valuation given to each variable (explicit as well as implicit) within the system. Consequently, the state shall be represented by the term θ, and will be understood to encapsulate all the current variable valuations. Note that $\preceq$ refers to the subsequence operator, whereas $\preceq_s$ refers to the subsequence suffix operator. Also, given sequence α, first(α) refers to the first element in α, last(α) refers to the last element in α and items(α) returns the set of elements found within α.

Definition 9.1.2 *The event trace is defined as*

$$EventTrace \stackrel{def}{=} seq\,(\,\mathbb{T} \times Event \times State\,)$$

which implies that the event trace is defined as a sequence of triples, where the Event denotes the basic event which has fired, the time denotes the time at which this event has fired (a timestamp), and the State denotes the underlying system state at the instant the event fired. Given that time has been defined as a non-negative real leads us to the following assumption

Assumption 9.1.1 *Given a pair* $(\,\mathbb{T} \times \mathbb{T}\,)$ *and an event trace E, the subsequence of E bounded by this pair must be of a finite length.*

This assumption, also known as that of finite variability (similarly to [8]), is imperative in order to avoid Zeno-like behaviour within our logic. Concisely, it dictates that given any subsequence of the event trace, this subsequence must contain a finite amount of event firings. This is equivalent to assuming that the limit of the timestamp associated with each element in the event trace is infinity. In other words, time never stops progressing indefinitely.

Given that the event trace type definition represents the set of all sequences of time, event and state triples, we can logically conclude that the type of an interval within our logic is the same as that of the event trace. In other words, since an interval is informally understood to be a subsequence of the event trace, then the type of the event trace and that of an interval must be the same.

Definition 9.1.3 *Given state* θ*, the boolean condition* $\mathcal{BC}$ *is defined as*

$$\mathcal{BC} \stackrel{def}{=} \theta \rightarrow \mathbb{B}$$

which implies that a boolean condition is a function from the current state θ to a boolean value.

Definition 9.1.4 *Given an event expression and a boolean condition, the point of interest is defined as*

$$POI \stackrel{def}{=} (\mathit{Event} \times \mathcal{BC})$$

formalising the notion of the point of interest motivated in section 5.2.1, whereby the point of interest is characterised by the event expression and corresponding boolean condition which are to be satisfied if the point of interest is to trigger.

Recall that using the notation defined within the DATE framework (see definition 3.2.2), basic event e fires composite event ev is written as $e \models ev$. Hence, the triggering of a point of interest is defined as

Definition 9.1.5 *Given a point of interest p, basic event e and state θ, the triggering of a point of interest is defined as follows*

$$\begin{array}{l} \mathit{trigger}_p\ e, \theta \stackrel{def}{=} (e \models ev) \wedge B(\theta) \\ \quad \mathit{where} \\ \quad\quad (ev,\ B) = p \end{array}$$

Given point of interest p, the occurrence of basic event e (an underlying system event, a channel event or a timer event) with current system state θ, p is said to fire if e fires event expression ev and boolean condition B when applied to current state θ returns true.

Definition 9.1.6 *Given an event trace es and time t', the absolute longest prefix prior to exceeding t is defined as follows*

$$\phi_a :: (\mathit{EventTrace} \times \mathbb{T}) \rightarrow \mathit{EventTrace}$$

$$\phi_a\ (t,\ e,\ \theta)\ :\ es\ ,\ t' \stackrel{def}{=} \begin{cases} \langle\ \rangle & \mathit{if}\ t \geq t' \\ (t,\ e,\ \theta)\ :\ \phi_a(es,\ t') & \mathit{otherwise} \end{cases}$$

In other words, ϕ_a returns the largest event trace subsequence whose duration is smaller or equal to t', assuming the first element in event trace es starts at time zero. We shall now extend the notion of the absolute longest prefix to allow for event trace sequences not starting from time zero.

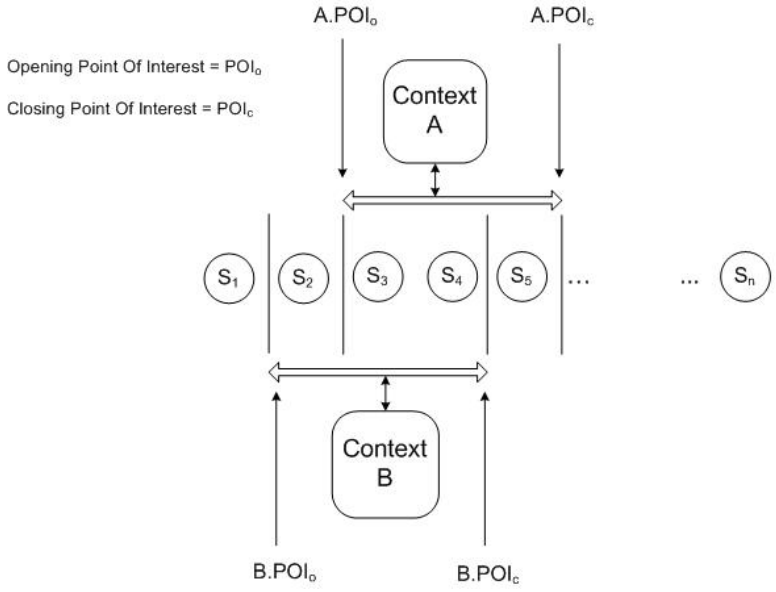
Definition 9.1.7 *Given an event trace es and time t', the longest prefix prior to exceeding t is defined as follows*

$$\phi :: (\mathit{EventTrace} \times \mathbb{T}) \rightarrow \mathit{EventTrace}$$

$$\phi\ (t,\ e,\ \theta)\ :\ es\ ,\ t' \stackrel{def}{=} \phi_a\ (\ (t,\ e,\ \theta)\ :\ es\ ,\ (\ t' + t\ )\ )$$

Informally, since the event trace starts at time t $\neq 0$, we shall extend time duration t' by t thus negating the effect of the event trace not starting at 0. Hence, the absolute longest prefix can be applied using the adjusted time duration. Note that the definition of the longest prefix justifies assumption 9.1.1, since the definition of the longest prefix requires that each progressively encountered timestamp in the sequence never indefinitely stops progressing.

Through the use of replication, we shall encode the issue of context for the representation of overlapping intervals as follows



Where each created object representing an interval context automatically generates its own set of events. Therefore, through the use of context we end up with a set of interval statistics, each listening to events bound solely to the associated interval context. Since the same interval statistic instance must either be or not be within an interval (cannot be both in the same time), using this representation we have reduced the issue of overlapping intervals into a 'family' of non-overlapping intervals, each listening on exclusive events bound to the associated context.

The definitions defined so far complete the basis for the remainder of the definitions of our formal semantics. Consequently, using these definitions, we shall now define each interval type in a straightforward fashion, starting from the event interval.

Definition 9.1.8 *The event interval is defined as follows*

$$EventInterval :: (POI \times POI \times EventTrace) \rightarrow 2^{EventTrace}$$

$$EventInterval\ p_o,\ p_c,\ es \stackrel{def}{=}$$
$$\{\ (\ \langle (t_1, o, \theta) \rangle\ +\!\!+\ es'\ +\!\!+\ \langle (t_2, c, \theta') \rangle \preceq es \mid trigger_{p_o}\ (\ o,\ \theta\ )\ \wedge trigger_{p_c}\ (\ c,\ \theta'\ ) \wedge$$
$$\forall\ (t_3, e, \theta'') \in items(\ es'\ ) \bullet\ \neg\ trigger_{p_c}(e\ ,\ \theta'')\ \}$$

The event interval is characterised by two points of interest (as motivated in section 5.4) – one triggering the opening of the interval and the other triggering the closing of the interval – and when applied to an event trace *es* returns a set of event traces representing the set of (possibly overlapping) intervals generated by the event interval. The crux of the event interval definition lies in the encoding of the statement that given the triggering of the opening, and the subsequent closing of the interval, there can be no element within this generated interval which triggers the closing of the interval. In other words, given the opening of an event interval, it will subsequently close at the first possible closing point of interest triggering.

Definition 9.1.9 *The duration interval is defined as follows*

$$DurationInterval :: (POI \times \mathbb{T} \times EventTrace) \rightarrow 2^{EventTrace}$$

$$DurationInterval\ p_o,\ t,\ es \stackrel{def}{=} \{\ \langle (t_1, o, \theta) \rangle \mathbin{+\!\!+} es' \preceq_s es \mid trigger_{p_o}\ (\ o,\ \theta\ ) \bullet \phi(\ \langle (t_1, o, \theta) \rangle \mathbin{+\!\!+} es',\ t\ )\ \}$$

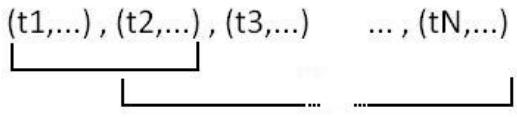
Whereby the duration interval is characterised by the opening point of interest p_o and a time duration t, and when applied to an event trace *es* returns the set of intervals generated by the duration interval. The duration interval generates intervals whose opening is triggered by the opening point of interest p_o, and closes on the last subsequent element prior to elapsing t. Using the longest prefix ϕ, the duration interval generates the largest possible subsequence starting from each element in the event trace triggering the opening interval point of interest. The subsequence suffix operator is hence used to ensure that each generated interval is the largest possible interval, thus avoiding the generation of all possible subsequences of intervals adhering to the duration interval definition.

Definition 9.1.10 *The time interval is defined as follows*

$$TimeInterval :: (\mathbb{T} \times EventTrace) \rightarrow 2^{EventTrace}$$

$$TimeInterval\ t,\ es \stackrel{def}{=} \{\ e \preceq_s es \bullet \phi(\ e,\ t\ )\ \}$$

Since the time interval is a dynamic interval 'moving' with time – as motivated in section 5.4 – this interval type dissects the event trace into intervals less than or equal to (in duration) t, where each resulting interval is considered to be a 'snapshot' of the event trace generated during the time interval movement, as exemplified below



Assuming that t3 - t1 > t and t3 - t2 ≤ t, the first time interval would have contained the first two elements (starting at times t1 and t2 respectively). However, as time passes t1 becomes

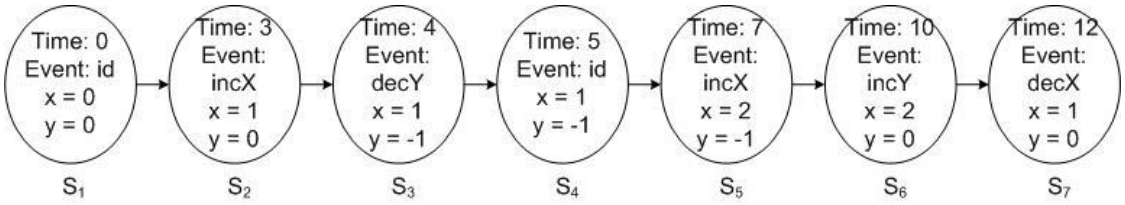
'outdated' and is hence discarded, giving rise to a new interval starting from the element whose occurrence is at time t2, and so on. The event trace dissection terminates with the termination of the event trace itself.

Example

The following is an example depicting the resulting intervals given an event trace and an interval definition. Assuming that the system state contains an evaluation for variables x and y, the set of possible events Ev = { incX, decX, incY, decY, id } and the event trace given a typical runtime execution of a program incrementing and decrementing the values of x and y is the following

$$\text{es} = \langle\ (0, \text{id},(0,0)),\ (3, \text{incX},(1,0)),\ (4, \text{decY},(1,\text{-}1)),\ (5, \text{id},(1,\text{-}1)),\ (7, \text{incX},(2,\text{-}1)),\ (10, \text{incY},(2,0)),\ (12, \text{decX},(1,0))\ \rangle$$

where system state (a,b) implies the following valuations x = a and y = b. This event trace can be visualised as



Below are a few example interval evaluations given the definitions above

EventInterval (incX, $(x - y \leq 1)$), (incY, $(y \geq 0)$), es $=$ { $\langle S_2, S_3, S_4, S_5, S_6 \rangle$ }

DurationInterval ((incY + decY), true) 2 es $=$ { $\langle S_3, S_4 \rangle$, $\langle S_6 \rangle$ }

TimeInterval 3 es $=$ { $\langle S_1, S_2 \rangle$, $\langle S_2, S_3, S_4 \rangle$, $\langle S_3, S_4, S_5 \rangle$, $\langle S_4, S_5 \rangle$, $\langle S_5, S_6 \rangle$, $\langle S_6, S_7 \rangle$ }

9.2 Formal Conversion

The following section formalises the techniques motivated in chapter 6 for the conversion of our statistical framework to constructs defined within the runtime verification framework using DATEs. Consequently, knowledge of DATEs (see section 3.2, [12]) is a prerequisite to this section. Moreover, since the statistical framework is 'embedded' within the RV framework using DATEs – as motivated in section 5.8 – the statistical framework will require information and also enrich structures used by the DATE framework, as will be evidenced below. Such structures include the number of declared channels and the underlying system state in order to store the

current statistic valuation within the state. In similar fashion to [35], this translation occurs in two main phases:

- **Pre-processing phase** — The phase which augments certain structures present within the RV framework with additional information required by the statistical framework prior to the actual conversion.
- **Statistic conversion phase** — The actual conversion of the defined statistic construct into a DATE.

Three statistical constructs have been broadly motivated in section **??**, these being the point and interval statistics (which additionally defines three interval types), as well as the statistical event. Consequently it is these three constructs which will be converted, starting from the conversion of the statistical events.

Pre-Processing Phase

Since statistical events are directly converted into DATE channels – as motivated in section 6.1 – the pre-processing phase is solely dedicated to creating a channel for each statistic defined, and adding these channels to the channels already defined within the RV framework. Note that for now we will assume *Channel* to be a type encapsulating all possible value passing channel definitions, while *Statistic* is a type containing all possible statistic declarations. Whereas the notion of a channel within the DATE framework has already been properly defined (section 3.2), we shall be refining the notion of a statistic declaration later on.

Definition 9.2.1 *Given the set of channels C declared within the DATE framework and the set of statistics S declared within the statistical framework*

$$C \ ::= \ C \ \cup \ \{ \, x \ \in \ S \ \bullet \ \nu \mathit{Channel} \, \}$$

where νChannel represents a fresh channel instance of type Channel. The definition above serves to give a clear picture regarding the underlying mechanism for the handling of statistical events, whereby each possible statistical event is emulated by a value passing channel – one for each statistic declaration. Later on we shall be using these fresh instances by passing the new statistical valuations upon the triggering of a statistic update.

Statistic Conversion Phase

The statistic conversion phase is responsible for converting the point and interval statistic constructs defined within the statistical framework into DATEs. Prior to these constructions we shall formalise a set of fundamental notions, starting from the notion of an action

Definition 9.2.2 *An action $\mathcal{A}$ is defined as*

$$\mathcal{A} \ \stackrel{def}{=} \ (\, \theta \ \rightarrow \ \theta \,) \ \times \ \mathcal{T}\mathcal{A}$$

which implies that the action (as motivated in section 5) is represented by a function altering the system state and a timer action altering the timer configuration (see section 3.2). This definition of $\mathcal{A}$ is a direct result of the distinction made in [12] between the action on the underlying system state and the action on the timer configuration. Therefore, since the statistical framework allows for actions on both the system as well as the timers, we require the definition of an action as being sufficiently expressive in order to alter both the system state as well as the timer configuration.

Definition 9.2.3 *Given state θ and timer configuration $\mathcal{CT}$, the augmented boolean condition $\mathcal{BC}_A$ is defined as*

$$\mathcal{BC}_A \stackrel{def}{=} (\theta \times \mathcal{CT}) \rightarrow \mathbb{B}$$

$\mathcal{BC}_A$ is a function from the current state θ and timer configuration $\mathcal{CT}$ to a boolean value. The requirement of this augmented boolean condition stems from the separation of the system state and the timer configuration within the DATE framework. Consequently, $\mathcal{BC}_A$ is defined since theoretically, a boolean condition within the statistical framework could be based on both system and timer information. This issue of separation of system state from timer configuration is in fact a recurring one throughout the rest of this section. In truth, the system state and timer configuration were considered as one global state throughout the previous section for simplicity. However, the concept of state can in fact be extended into the system state and timer configuration without affecting the previous results. Since the DATE framework defines all constructs with this augmented concept of state, we shall henceforth be aligning our concept of state with that of the DATE framework.

Definition 9.2.4 *Given an event expression and an augmented boolean condition, the augmented point of interest is defined as*

$$POI_A \stackrel{def}{=} (\ Event \times \mathcal{BC}_A\ )$$

whereby the augmented point of interest has an identical definition to definition 9.1.4, apart from the definition of the augmented boolean condition.

Definition 9.2.5 *Given an augmented point of interest p, basic event e, state θ and timer configuration $\mathcal{CT}$, the triggering of an augmented point of interest is defined as follows*

$$\begin{aligned} trigger_p\ e, \theta, \mathcal{CT} \stackrel{def}{=}\ & (\ e \models ev\ ) \ \wedge\ B(\ \theta, \mathcal{CT}\ ) \\ & where \\ & \quad (ev,\ B) = p \end{aligned}$$

Again the semantics of the triggering of the augmented point of interest is identical to previous definition 9.1.5, with the definition augmented to handle augmented points of interest. Consequently, an additional parameter is accepted declaring the current timer configuration. Note that the term 'augmented' will be dropped for brevity. However, keep in mind that henceforth

whenever referring to a boolean condition or point of interest, we imply that we are referring to the augmented versions.

Definition 9.2.6 *Given a point of interest and an action, an interval point of interest is defined as*

$$IPOI \stackrel{def}{=} (\, POI_A \times \mathcal{A} \,)$$

which formalises the notion of an interval point of interest presented in 5.4, whereby an interval point of interest is simply a point of interest with an added action attached.

The statistic configuration is essentially the current statistic valuation, and in the case of interval statistics whose interval type is either an event or duration interval, the state whether the statistic is within or outside an interval must also be taken into consideration. However, while the statistic state (as will be seen below) will be encoded within the DATE automaton configuration, there is a need to develop a mechanism which represents the current statistic valuation. We shall be taking a different approach to that taken for timer configurations in that we shall be assuming that all statistic valuations are stored within the underlying system state θ — much like the DATE framework assumes that any declared automaton variables are embedded within the system state. This design choice aids in simplicity since it allows for a straightforward representation of the fact that action $\mathcal{A}$ can also alter statistic valuations. Also, using the above definition of $\mathcal{A}$ allows for the actions on statistics (motivated in section 5.5) to be encoded as actions in a straightforward fashion. With the current statistic valuations stored within the system state, we require a mechanism with which to extract the statistic valuation for a particular statistic. Consequently, auxiliary function statConf will be assumed.

Definition 9.2.7 *Given an id representing the statistic identifier and the current system state , statConf returns the current statistic valuation of type X*

$$statConf :: (\, \mathcal{ID} \times \theta \,) \rightarrow X$$

Since a statistic type could theoretically differ (such as being of type integer or real), we shall be referring to the statistic type as the general type X. This implies that type X could be changed to any required type without affecting the following results. Typical types include $\mathbb{I}$ and $\mathbb{R}$. As will be presented below, each statistic is defined with an associated id (its identifier), and it is through this identifier that statConf is able to extract the current statistic valuation from θ.

Construction 9.2.1 *Given an identifier id, point of interest p, initial statistic valuation* v_0 *statistic update u, system state* θ *and initial system state* θ_0 *the construction converting the point statistic is defined as*

$$PointStat :: (\mathcal{ID} \times POI_A \times X \times \mathcal{A}) \rightarrow (DATE \times DATEConf_0)$$

$$\begin{aligned} PointStat\, id,\, p,\, v_0,\, u \stackrel{def}{=}\ & \langle \{ M \}, \{\} \rangle \wedge conf_0 \\ & where \\ & (e, b) = p \\ & (u_s, u_t) = u \\ & M = \langle \{ q_0 \}, q_0, \rightarrow, \{\}, \{\} \rangle \\ & \rightarrow = \{ (q_0, e, b, u_t, id(v), u_s, q_0) \} \\ & v = statConf(id, \theta) \\ & conf_0 = (\{\}, \theta_0 \cup v_0, q_0) \end{aligned}$$

The point statistic is characterised by (i) an identifier (ii) an augmented point of interest specifying when to trigger the statistic update (iii) a value of type X representing the initial statistic value (iv) an action representing the statistic update. In return, the construction returns (i) a DATE and (ii) the initial DATE configuration.

Recall the point statistic conversion specified in section 6.2, as summarised in the following diagram

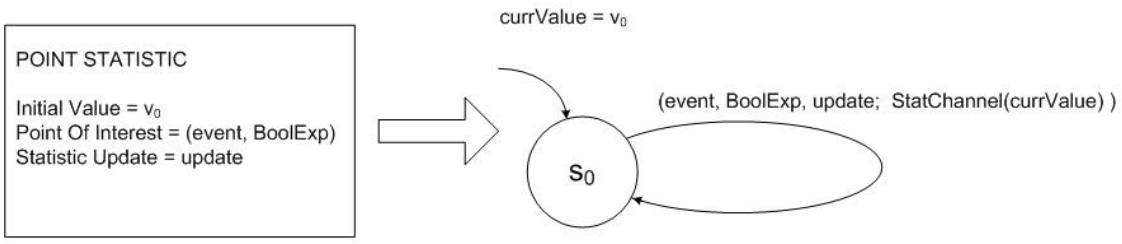


Figure 9.1: Point Statistic Conversion

Construction 9.2.1 formalises the above translation. Hence, the result is a one-state DATE lacking an automaton constructor definition (since point statistics have no notion of dynamic replication). Also notice how the resulting DATE lacks any accepting or rejecting states, since the point statistic has no such notion of acceptance or rejection, but only of evaluation. Evaluation is defined through the transition, which is triggered upon the satisfaction of the event expression and augmented boolean condition – the point of interest components – and executes both the action on the state and the timer actions. Note that the current statistic evaluation is sent over channel id (generated during the pre-processing phase) upon the completion of the transition triggering, thus emulating the statistical event through this channel transmission. Also note that the actual execution of the statistic update is executed through the execution of the DATE step (Definition 3.2.10).

The initial DATE configuration is defined as lacking an initial timer configuration (since point statistics require no clocks), with the addition of the initial statistic value v_0 with the initial system state θ, and the initial automaton state being q_0. The addition of v_0 with θ serves to formalise the design choice previously discussed of adding statistic valuations to the system state.

Definition 9.2.8 *An interval is defined as*

$$Interval \stackrel{def}{=} (IPOI \times IPOI) \mid (IPOI \times \mathbb{T} \times \mathcal{A}) \mid (\mathbb{T} \times \mathcal{A})$$

Whereby an interval is characterised either (i) by two interval points of interest – an event interval (ii) by a point of interest, a time duration and an action – a duration interval, or (iii) by a time duration and an action – a time interval. The notion of the interval forms the backbone of the interval statistic.

Definition 9.2.9 *The signature characterising the conversion of the interval statistic is defined as follows*

$$IntervalStat :: (\mathcal{ID} \times POI_A \times X \times Interval \times \mathcal{A}) \rightarrow (DATE \times DATEConf_0)$$

Whereby the interval statistic is characterised by (i) an identifier (ii) an augmented point of interest specifying when to trigger the statistic update (iii) a value of type X representing the initial statistic value (iv) an interval definition (v) an action representing the statistic update. In return, the construction returns (i) a DATE and (ii) the initial DATE configuration. As is evident, there exists a similarity between the interval statistic signature and that of the point statistic – the only addition to the interval statistic is the definition of the interval – which substantiates the theory that a point statistic is an unbounded interval statistic, or conversely that an interval statistic is a point statistic over a set of event trace subsequences dictated by the interval.

As described in chapter 6, the interval statistic conversion depends on the statistic's interval type. Consequently, we shall present three distinct constructions, one for each interval type, starting from the construction for the conversion of an interval statistic whose interval type is an event interval (henceforth event interval statistic for brevity).

Construction 9.2.2 *Given an identifier id, point of interest p, initial statistic valuation v_0, event interval $\mathcal{E}$, statistic update u, system state θ and initial system state θ_0 the construction converting the event interval statistic is defined as*

$$
\begin{array}{rcl}
\mathit{IntervalStat}\, id,\, p,\, v_0,\, \mathcal{E},\, u & \stackrel{def}{=} & \langle\, \{\, M \,\},\, \{\} \,\rangle \,\wedge\, \mathit{conf}_0 \\
\multicolumn{3}{c}{\textit{where}} \\
(\, e,\, b \,) & = & p \\
(\, u_s,\, u_t \,) & = & u \\
(\,(\,(\, e_o,\, b_o \,),\, (\, u_{os},\, u_{ot} \,)),\, ((\, e_c,\, b_c \,),\, (\, u_{cs},\, u_{ct} \,))\,) & = & \mathcal{E} \\
M & = & \langle\, \{\, q_0,\, q_1 \,\},\, q_0,\, \rightarrow,\, \{\},\, \{\} \,\rangle \\
\rightarrow & = & \{\, (\, q_0,\, e_o,\, b_o,\, u_{ot},\, \{\},\, u_{os},\, q_1 \,), \\
 & & \quad (\, q_1,\, e,\, b,\, u_t,\, \mathit{id}(v),\, u_s,\, q_1 \,), \\
 & & \quad (\, q_1,\, e_c,\, b_c,\, u_{ct},\, \{\},\, u_{cs},\, q_0 \,)\, \} \\
v & = & \mathit{statConf}(\, id,\, \theta \,) \\
\mathit{conf}_0 & = & (\, \{\},\, \theta_0 \cup v_0,\, q_0 \,)
\end{array}
$$

The following diagram (taken from section section 6.3) summarises the translation of the event interval statistic into a DATE construct.

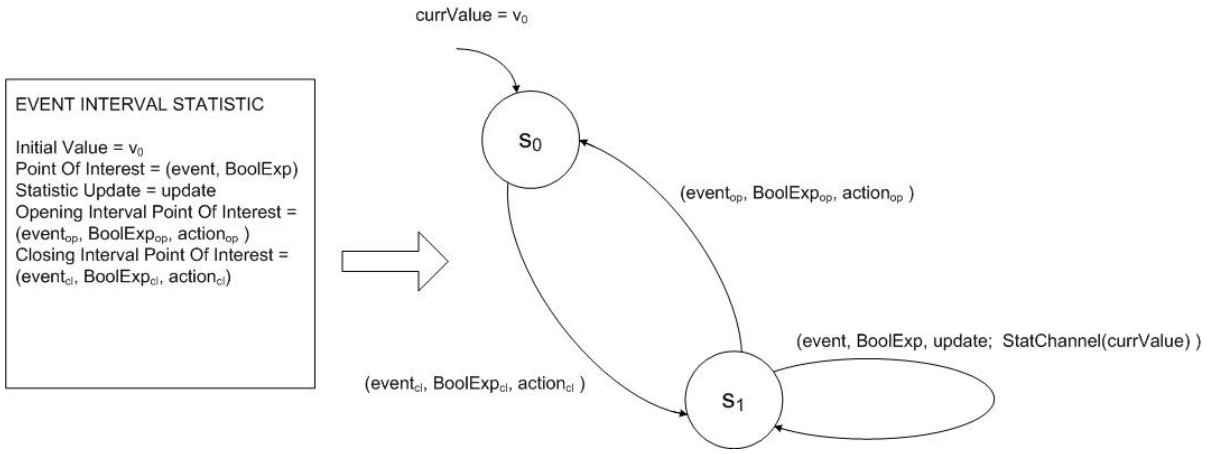


Figure 9.2: Event Interval Statistic Conversion

The resulting DATE (lacking an automaton constructor definition) is a two-state DATE, with the initial state representing the statistic state when being not being within an interval (q_0) and another state representing the statistic being within an interval (q_1). Three transitions are defined, one which alters the statistic state from q_0 to q_1 thus opening an interval, another from q_1 to q_1 firing the statistic update (since the statistic is within the interval), and another from q_1 to q_0 closing the interval. Therefore, the first transition fires upon the triggering of the opening interval point of interest and executes the associated action, the second transition fires upon the triggering of the point of interest and updates the statistic valuation through the execution of the statistic update, and the third transition fires upon the triggering of the closing interval point of interest, and executes the associated action. Note that upon the firing of the

second transition, the new statistic valuation is sent on the associated channel created during the pre-processing phase.

The initial DATE configuration is defined as lacking an initial timer configuration (event intervals require no clocks), with the addition of the initial statistic value v_0 with the initial system state θ, and the initial automaton state being q_0.

Construction 9.2.3 *Given an identifier id, point of interest p, initial statistic valuation v_0, duration interval $\mathcal{D}$ statistic update u, system state θ, initial system state θ_0 and timer t_0 the construction converting the duration interval statistic is defined as*

$$
\begin{array}{rcl}
\mathit{IntervalStat}\ id,\ p,\ v_0,\ \mathcal{D},\ u & \stackrel{def}{=} & \langle\, \{\, M \,\},\ \{\} \,\rangle \wedge \mathit{conf}_0 \\
\textit{where} & & \\
(\, e,\ b \,) & = & p \\
(\, u_s,\ u_t \,) & = & u \\
(\,(\,(\, e_o,\ b_o \,),\ (\, u_{os},\ u_{ot} \,)\,),\ \lambda,\ (\, u_{ds},\ u_{dt} \,)\,) & = & \mathcal{D} \\
M & = & \langle\, \{\, q_0,\ q_1 \,\},\ q_0,\ \rightarrow,\ \{\}\ \{\} \,\rangle \\
\rightarrow & = & \{\, (\, q_0,\ e_o,\ b_o,\ t_0\mathit{Reset} \circ u_{ot},\ \{\},\ u_{os},\ q_1 \,), \\
 & & \quad (\, q_1,\ e,\ b,\ u_t,\ \mathit{id}(v),\ u_s,\ q_1 \,), \\
 & & \quad (\, q_1,\ t_0 \mathbin{@} \lambda,\ \mathit{true},\ u_{dt},\ \{\},\ u_{ds},\ q_0 \,)\ \} \\
v & = & \mathit{statConf}(\, id,\ \theta \,) \\
\mathit{conf}_0 & = & (\,(\, t_0,\ 0,\ \mathit{running} \,),\ \theta_0 \cup v_0,\ q_0 \,)
\end{array}
$$

The following diagram (taken from section section 6.3) summarises the translation of the duration interval statistic into a DATE construct.

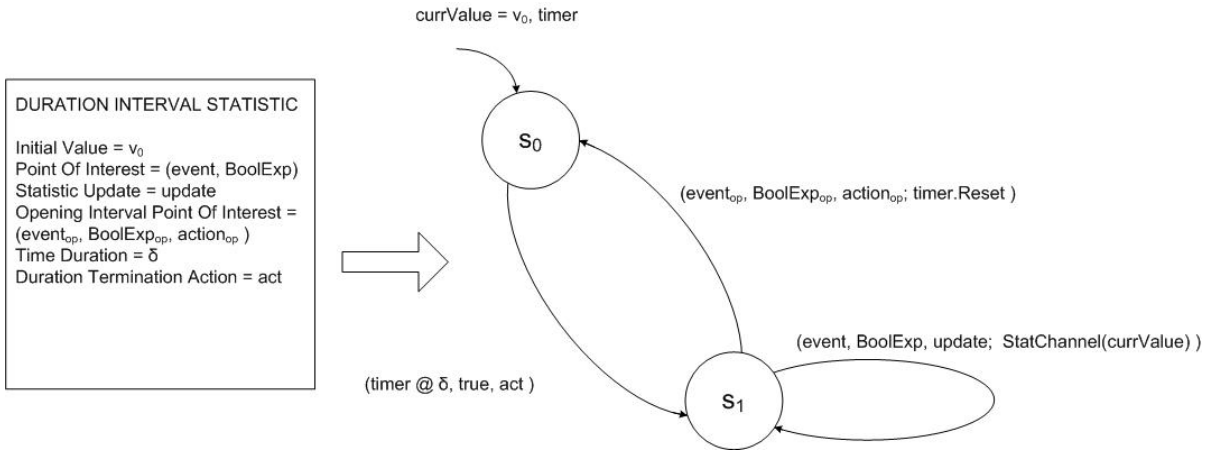


Figure 9.3: Duration Interval Statistic Conversion

The approach taken for the conversion of the duration interval statistic is identical to that taken for the conversion of the event interval statistic, with the addition of timer t_0. The alterations include the resetting of t_0 upon entering the within-interval state (represented by timer action t_0Reset which resets the value of t_0 to 0), the firing of the transition closing the

interval after elapsing λ time units (the interval duration) and finally the initial configuration which declares the initial timer configuration for t_0 with an initial time of 0 and in the running state. Hence, at any instance timer configuration $\mathcal{CT}$ contains the configuration for t_0.

Construction 9.2.4 *Given an identifier id, point of interest p, initial statistic valuation v_0, time interval $\mathcal{T}$ statistic update u, system state θ, initial system state θ_0, timer t_0 and polling interval δ the construction converting the interval statistic is defined as*

$$
\begin{array}{rcl}
\mathit{IntervalStat}\ id,\ p,\ v_0,\ \mathcal{T},\ u & \stackrel{def}{=} & \langle\, \{\, M \,\},\, \{\} \,\rangle \wedge \mathit{conf}_0 \\
\quad \mathit{where} & & \\
(\, e,\, b \,) & = & p \\
(\, u_s,\, u_t \,) & = & u \\
(\, \lambda,\, (\, u_{ps},\, u_{pt} \,) \,) & = & \mathcal{T} \\
M & = & \langle\, \{\, q_0 \}, \ q_0, \ \rightarrow, \ \{\} \ \{\} \,\rangle \\
\rightarrow & = & \{\, (\, q_0,\ t_0 \mathbin{@} \delta,\ \mathit{true},\ t_0 \mathit{Reset} \circ u_{pt},\ \{\},\ u_{ps},\ q_0 \,), \\
 & & \quad (\, q_0,\ e,\ b,\ u_t,\ \mathit{id}(v),\ u_s,\ q_0 \,) \,\} \\
v & = & \mathit{statConf}(\, id,\, \theta \,) \\
\mathit{conf}_0 & = & (\, (\, t_0,\, 0,\, \mathit{running} \,),\, \theta_0 \cup v_0,\, q_0 \,)
\end{array}
$$

The following diagram (extraction from section section 6.3) summarises the translation of the time interval statistic into a DATE construct.

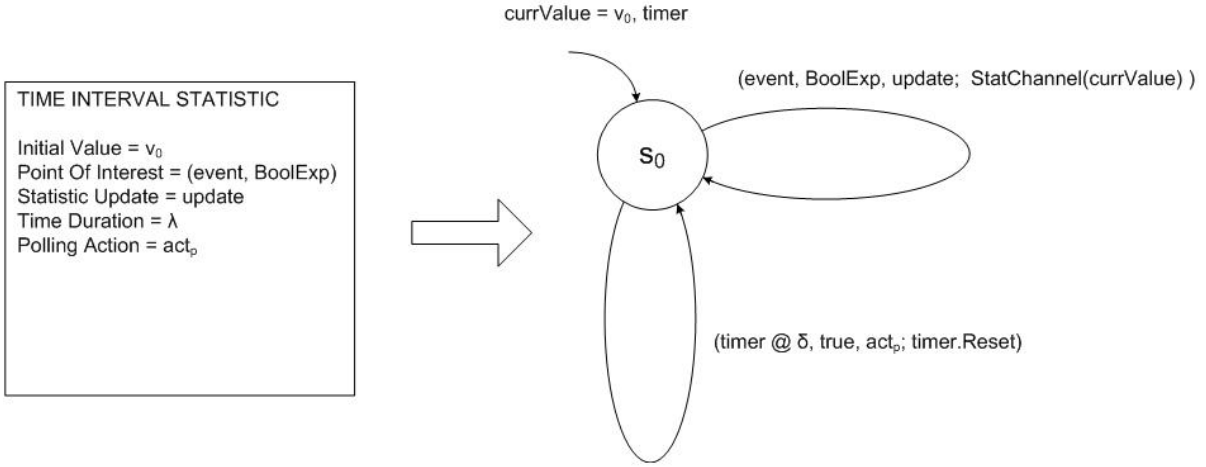


Figure 9.4: Time Interval Statistic Conversion

The time interval statistic construction is somewhat different than the other two interval statistic constructions. The result is a one-state DATE with two transitions, one which emulates the polling mechanism (motivated in section 5.4) and another which handles statistic updates. Note the use of δ within the definition of the first transition, whereby t_0 is firing the transition every δ time units. Upon the firing of this transition, the polling update action is executed so as to keep the transition valuation in line with the time interval semantics. Also note the lack of use of λ (the time interval duration) within the construction. In practice the

use of λ is embedded within u_{ps} and u_{pt} (the polling update action components), as explained in section 5.4. The initial DATE configuration is identical to that defined for the duration interval statistic conversion, whereby t_0 is initialised and set to running, v_0 is added to initial system state θ_0 and the initial automaton state is set to q_0.

On a concluding note, we shall assume that an ordering on the statistic declarations is present within our statistical framework representing the statistic ordering defined in section 5.6. Moreover, since the declared statistics are converted into DATEs, we shall assume that this statistic ordering is translated into the ordering on automata present within the DATE framework as defined in [12].

9.3 Conclusions

This chapter presented a formalisation of the core notions driving the statistical framework, more specifically the notions of the interval as well as the conversion of the statistical constructs to DATEs. Hence, this chapter formally specifies the operational semantics given to the logic defined for the expression of statistics collection. The conversion formalisation was rather straightforward due to following the substantial motivation discussed in chapter 6, as well as the fact that the logic was design with the straightforward connection between the statistical and runtime verification frameworks kept in mind. Although the representation of context within our formalisation is considered sufficient, such a crucial issue could have benefitted a more complete formal representation. In truth, we opted against this idea in order to keep the rest of the formal definitions as straightforward as possible.

Finally, it is worth pointing out that for completeness' sake one would ideally link both formalised notions, whereby one would have to prove that the DATEs (representing statistics) open, close and update at the exact instances dictated by the interval semantics. However, this proof will not be attempted since it would be rather long and tedious, while not being particularly revealing.

10. Case Studies

This chapter focuses on introducing the practical side of LarvaStat through the discussion of a few select real life scenarios. Hence, we shall be presenting a few case studies applied on an application chosen as the underlying system. Note that this chapter also serves as an introductory chapter to the next, whereby a comprehensive case study involving intrusion detection, system profiling as well as probabilistic monitoring is discussed. Crucially, keep in mind that all developed case studies within both chapters specify additional functionality through the specification of a concise LarvaStat script, without actually modifying a single line of original system code. Hence, this implies that complex additional functionality through the specification of statistics can be automatically integrated with the underlying system through the use of LarvaStat scripts. In truth, apart from applying the framework on a real life scenarios, the specification of the following case studies presented an excellent platform as an aid for our statistical framework in reaching a certain maturity. In fact, the following case studies were crucial in recognising certain issues with the framework which were later rectified, such as the issue of overlapping intervals as well as the handling of statistical events across contexts.

The following section gives a brief overview of the application chosen as the underlying system for the case studies. This is followed by a section introducing a few concise statistical property specifications applied to this application, which is henceforth followed by the presentation of three more comprehensive scenarios. The last section concludes the chapter.

10.1 The ftpd server

A Java implementation of a typical ftpd[1] server has been chosen as the underlying system for the following case studies presented in this as well as the following chapters. This particular Java implementation is called the Anomic FTP server[2], freely available for redistribution and modification under the GNU General Public Licence. Moreover, this implementation adheres to the standard RFC 959 protocol [23] specifying the File Transfer Protocol (FTP), which makes this ftpd server implementation accessible to most web browsers and ftp clients.

[1] the 'd' at the end stands for daemon, and is inherited from the name assigned to the Unix ftp server implementation.

[2] Available at http://www.anomic.de/AnomicFTPServer/index.html

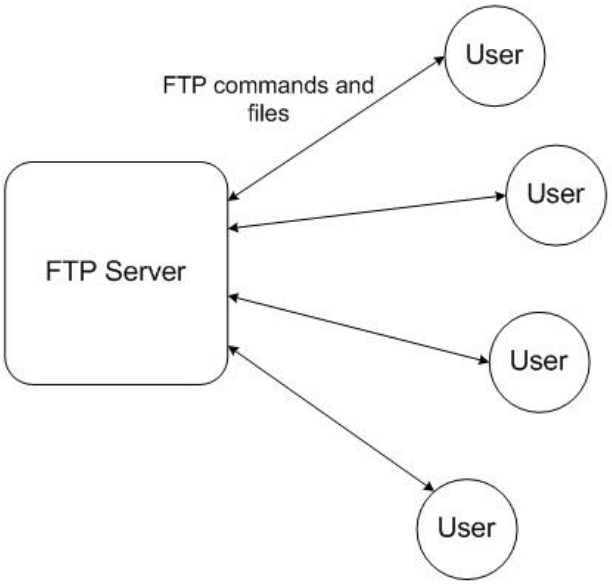


Figure 10.1: ftpd Server Overview

An ftpd server is essentially an application which, in conjunction with an ftp client, allows users to remotely manipulate files located on the server. Such manipulation includes downloading and uploading of files, as well as the alteration of files on the server. The FTP protocol runs on the network standard Transmission Control Protocol (TCP), usually through port 21, and specifies a set of commands which the ftp client can send to the server for communication. Hence, the client issues FTP commands to the server, and the server replies accordingly possibly by transferring files back to the client.

Note that the Anomic FTP server encapsulates a suitable representation of an FTP server, since not only does it implement all the FTP commands specified by the RFC 959 protocol [23], but also extends this functionality. In fact, this ftp server implements extensive user management functionality (allowing the specification of users and user groups, with each group specifying a set of rights and home directory), a built in firewall as well as remote administration.

From the point of view of the specification of statistical properties, choosing this program as our underlying system offers a wealth of interesting applications. Such applications may include the specification of properties on user behaviour (thus allowing for user profiling — also allowing for the specification of intrusion detection systems), the system performance through the analysis of packet rate retransmission as well as the analysis of other issues at the network level (performance profiling), and also statistical properties regarding the implemented firewall. Also importantly is the fact that the server is modularly implemented, which allows for an intuitive specification for most required specifications. Moreover, this server implementation allows for multiple user connections, as well as multiple concurrent downloads. Hence, features within our framework such as overlapping intervals can be extensively used within the context of this system.

The following is a broad overview of the classes of interest within the server implementation required during the specification of statistical properties.

- serverCore — The core implementation class responsible for the handling of connecions, as well as the execution of the requested FTP commands.
- Session — An internal object representing the session created by the acceptance of a user connection. Hence, the session object contains information regarding the client, most notably such as the client IP address, as well as the input and output streams responsible for communication between the client and the server
- ftpdProtocol — The class implementing all FTP commands. Hence, upon transmission of a command by the client, the server passes execution to an instance of this class for the execution of the command. Note that each session contains an instance of this class.
- ftpdPermissions — The class responsible for checking that the particular client has the required permissions for the execution of certain commands.

For more information regarding the server implementation, as well as a user manual please visit the official website.

10.2 A Few Simpler Examples

The following section introduces a few simpler case study scenarios using the ftpd server as the underlying system.

Case Study 1 — The Bad Login Count

We require the additional logic of terminating a session connection if three consecutive bad logins occur within the same session. This is achieved through the implementation of a single point statistic keeping count of the number of successive bad login events, and a property which reacts to the valuation of this point statistic accordingly

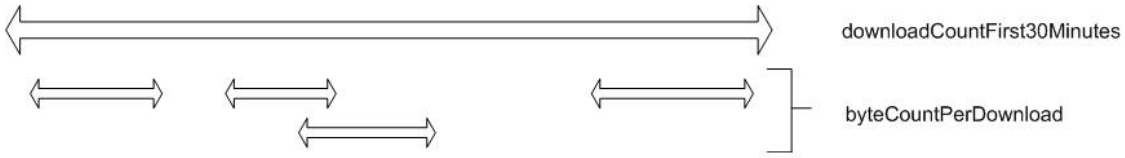
```
GLOBAL  {
  FOREACH (Session s)  {
          EVENTS  {
               passwordEvent(ftpdProtocol p,String res) =
           {p.PASS(arg1) uponReturning (res) } where {s = p.getSession();}
    }

    POINTSTAT BadLoginCount : Integer  {
      INIT{BadLoginCount.setValue(new Integer(0));}

      EVENTS{passwordEvent}

      CONDITION{res.equals("530 authorization failed.")}
```

```
      UPDATE{ BadLoginCount.setValue(BadLoginCount.getValue() + 1); }
    }

    PROPERTY monitorCount  {
      STATES  {
        BAD { }  NORMAL { }  STARTING { start }
      }
      TRANSITIONS  {
        start -> start[passwordEvent\ \
                       if (res.equals("230 logged in")
                           BadLoginCount.setValue(new Integer(0));  ]
        start -> start[BadLoginCount_Event\ \
                       if (BadLoginCount_Value > 2) {  s.close(); }  ]
      }
    }
  }
}
```

Point statistic BadLoginCount keeps count of the number of times event passwordEvent reports failure. Note that since this event is defined within the Session context, each context will only be notified of password events specific to that context. Property monitorCount is responsible for two issues; resetting the point statistic valuation if the password event reports success, or terminates the session if the statistic valuation exceeds two. Hence, the first defined transition listens to the password event, executing a reset action on the statistic object representing BadLoginCount, whereas the second transition listens on BadLoginCount's statistical event, and closes the connection if the statistic valuation passed on the event exceeds two.

Case Study 2 — Downloads within First Thirty Minutes

We would like to express a statistic which collects the byte count downloaded within the first thirty minutes since login for each session. This statistic is expressed as a two-layer statistic (within the same context), whereby an overlapping event interval statistic collects the amount of bytes downloaded for each download, and another duration interval statistic collects the total amount of bytes downloaded since login for the first thirty minutes, as expressed below



Using the statistical event functionality, we can specify this nested multilayered statistic functionality. Hence, duration interval statistic downloadCountFirst30Minutes listens on the closing interval statistical event of event interval statistic byteCountPerDownload, adding the value sent of the statistical event to the count marking the total download count within the first thirty minutes since login.

```
GLOBAL  {
  EVENTS {
```

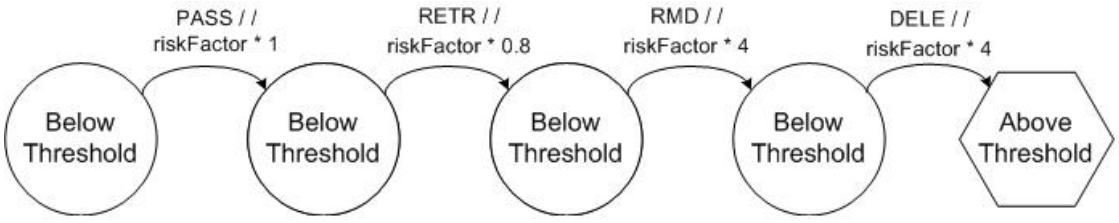
```
  sendInfo(OutputStream out) = {call out.write(arg1,arg2,arg3) }
}

FOREACH (Session s)  {
  EVENTS  {
    passwordEvent(String res) = {execution ftpdProtocol p.PASS(arg1)
                                  uponReturning (res) } where { s = p.getSession(); }
    downloadStarting(ftpdProtocol p) = {call p.eventDownloadFilePre(arg1)}
                                                        where { s = p.getSession(); }
    downloadComplete(ftpdProtocol p) = {call p.eventDownloadFilePost(arg1,arg2)}
                                                        where { s = p.getSession(); }
  }

  INTERVALSTAT byteCountPerDownload : Integer  {
    INIT{byteCountPerDownload.setValue(new Integer(0));}

    EVENTS{sendInfo}

    CONDITION{ out == s.out }

    LIST{ false }

    INTERVAL [CONTEXT (ftpdProtocol ftpdP)]
    {
      OPEN [downloadStarting \ \ ] WHERE {p == ftpdP}
      CLOSE [downloadComplete \ \ ] WHERE {p == ftpdP}
    }   OBJECTCONTEXT { s = ftpdP.getSession(); }

    UPDATE{ byteCountPerDownload.setValue(
                byteCountPerDownload.getValue() + ftpdP.bufferSize); }
  }

  INTERVALSTAT downloadCountFirst30Minutes : Integer  {
    INIT{downloadCountFirst30Minutes.setValue(new Integer(0));}

    EVENTS{byteCountPerDownload_EventComplete}

    LIST{ false }

    INTERVAL
    {
      OPEN [passwordEvent \ res.equals("230 logged in")  \ ]
      DURATION [1800 \ byteCountPerDownload.switchOff();]
    }

    UPDATE{downloadCountFirst30Minutes.setValue(
               downloadCountFirst30Minutes.getValue() +
               byteCountPerDownload_FinalValue);}
  }
}
```

```
}
```

Note that event sendInfo does not admit a direct manner with which to extract the session context (unlike the other three events). Nevertheless, although sendInfo is declared within the global context, it can still be referred to by constructs within sub contexts, such as statistic byteCountPerDownload, whereby the context is indirectly handled through the use of a condition (acting as a filter). Also note that whereas statistic byteCountPerDownload defines an interval with an associated context (since multiple concurrent downloads are allowed within the server), statistic downloadCountFirst30Minutes defines an interval lacking context (since each session admits only one good login). The interval context object defined for byteCountPerDownload is defined to be an instance of ftpdProtocol, since each Session object is assigned an instance of this class, and the actual events marking the opening and closing of the interval are executed by ftpdProtocol objects. Upon elapsing 1800 seconds (30 minutes), duration interval statistic downloadCountFirst30Minutes closes its interval, while also switching off byteCountPerDownload since no more byte counts per download are required.

Case Study 3 — The Action Suspicion

Each command defined within the FTP protocol is marked with an associated suspicion rating which approximately signifies the risk (to the system) associated with the execution of that command. If this rating is > 1, this implies that the command imposes a certain risk to the system, whereas if this rating is < 1 the command is ensured not to impose any risk (if the rating $= 1$ then the risk factor for the command is unknown, or neutral). Hence, given that client behaviour can be conceptualised as a sequence of command requests, using these suspicion ratings we can approximate a valuation representing the degree of risk implied by the client. If the client's risk factor exceeds a certain threshold, the user is banned from the system. As can be seen below, implementing this logic involves simulating a markov chain. Moreover, the implementation of this case study in fact represents the first approach to statistical anomaly intrusion detection using markov models (as discussed in [7]). Statistical anomaly detection shall be discussed in further detail in the following chapter.



The following LarvaStat script implements the required logic for the action suspicion case study.

```
GLOBAL  {
  FOREACH (Session s)  {
    VARIABLES  {
```

```
  Channel c = new Channel();
}

EVENTS  {
  passwordEvent(ftpdProtocol p) = {p.PASS(arg1)} where {s = p.getSession();}
  ChangeWorkingDirectoryEvent(ftpdProtocol p) = {p.CWD(arg1)}
                                                where {s = p.getSession();}
  downloadFileEvent(ftpdProtocol p) = {p.RETR(arg1)} where {s = p.getSession();}
  removeDirectoryEvent(ftpdProtocol p) = {p.RMD(arg1)} where {s = p.getSession();}
  deleteFileEvent(ftpdProtocol p) = {p.DELE(arg1)} where {s = p.getSession();}

  receiveValue(ObjectPair obj) = { c.receive(obj) }
                                                where {s = (Session)obj.getObject1();}
}

POINTSTAT ActionSuspicion : Double  {
  INIT{ActionSuspicion.setValue(new Double(1));}

  EVENTS{receiveValue}

  UPDATE{ ActionSuspicion.setValue(ActionSuspicion.getValue()
                                                * ((Double)obj.getObject2())); }
}

PROPERTY sendRiskProbability  {
  STATES  {
    BAD { }  NORMAL { }  STARTING { start }
      }
  TRANSITIONS  {
    start -> start[passwordEvent \  \ c.send(new ObjectPair(s,new Double(1))); ]
    start -> start[ChangeWorkingDirectoryEvent \  \
                                    c.send(new ObjectPair(s,new Double(0.5))); ]
    start -> start[downloadFileEvent \  \
                                    c.send(new ObjectPair(s,new Double(0.8))); ]
    start -> start[removeDirectoryEvent \  \
                                    c.send(new ObjectPair(s,new Double(4))); ]
    start -> start[deleteFileEvent \  \
                                    c.send(new ObjectPair(s,new Double(4))); ]
  }
}

PROPERTY isThresholdExceeded  {
  STATES  {
    BAD { }  NORMAL { }  STARTING { start }
  }
  TRANSITIONS
  {
    start -> start[ActionSuspicion_Event\ \
                    if (ActionSuspicion.getValue().doubleValue() > 10) {
                        s.close(); } } ]
```

```
      }
    }
  }
}
```

For brevity, certain FTP commands have been left out. Also note that the action suspicion for each command was rather arbitrarily chosen, mostly based on information collected from the RFC959 protocol. In truth, action suspicion values are usually chosen by system security officers with real life experience on such matters. Nevertheless, the implemented logic remains valid, with only certain constants possibly requiring alterations. Point statistic ActionSuspicion keeps track of the action suspicion valuation by multiplying all valuations sent on channel c. Property sendRiskProbability simply listens for the execution of FTP commands, and sends suspicion values accordingly. Property isThresholdExceeded listens on ActionSuspicion's statistical event, and terminates the session connection if the statistic valuation sent on the statistical event exceeds the (arbitrarily chosen) valuation of 10.

10.3 System Diagnostics

The current ftpd server does not currently implement any functionality for the extraction system diagnostics from the running server execution. Such diagnostics are vital for issues such as performance profiling as well as serving as an indication of possible intrusion. Hence, a set of statistics are defined collecting information which aids in giving a more clear picture regarding the state of the server execution. These diagnostics involve collecting the total number of user logins, the total number of downloads, as well as evaluating the largest amount of bytes downloaded by a session (possibly hinting at situations such as denial of service attack). The following is a statistic definition used throughout the declaration of the LarvaStat script for the implementation of the required system diagnostics.

```
POINTSTAT EventCount : Integer {
  INIT{#EventCount#.setValue(new Integer(0));}
  UPDATE{ #EventCount#.setValue(#EventCount#.getValue() + 1); }
}
```

Statistic declaration EventCount, defined in the package PointStatPackage, defines the nature of a point statistic whose nature is to keep count of an event. Note the use of tags which are eventually replaced with the statistic name instantiating the statistic declaration.

```
STATDEFINITIONS{  PointStatPackage.txt;  }
GLOBAL  {
  EVENTS  {
    sendInfo(OutputStream out) = {call out.write(arg1,arg2,arg3) }
    passwordEvent(String res) = {execution ftpdProtocol p.PASS(arg1)
                                                          uponReturning (res) }
    downloadStarting(ftpdProtocol p) = {call p.eventDownloadFilePre(arg1)}
    downloadComplete(ftpdProtocol p) = {call p.eventDownloadFilePost(arg1,arg2)}
```

```
  }

  POINTSTAT loginCount = PointStatPackage.EventCount[passwordEvent \
                                                (res.equals("230 logged in")]
  POINTSTAT downloadCount = PointStatPackage.EventCount[downloadComplete \]

  POINTSTAT largestDownloadCount : Integer  {
    INIT{largestDownloadCount.setValue(new Integer(0));}

    EVENTS{totalBytesDownloadedPerSession_Event}

    UPDATE{ if (largestDownloadCount.getValue() <
                totalBytesDownloadedPerSession_Value)
               largestDownloadCount.setValue(
                         totalBytesDownloadedPerSession_Value);  }
  }

  FOREACH (Session s)  {
    INTERVALSTAT BytesDownloadedCountPerDownload : Integer
    {
      INIT{BytesDownloadedCountPerDownload.setValue(new Integer(0));}

      EVENTS{sendInfo}

      CONDITION{ out == ftpdP.bOut }

      LIST{false}

      INTERVAL [CONTEXT (ftpdProtocol ftpdP)]{
        OPEN[downloadStarting \ \ ] WHERE {p == ftpdP}
        CLOSE[downloadComplete \ \ ] WHERE {p == ftpdP}
      }   OBJECTCONTEXT { s = ftpdP.getSession(); }

      UPDATE{ BytesDownloadedCountPerDownload.setValue(
                BytesDownloadedCountPerDownload.getValue() + ftpdP.bufferSize); }
    }

    POINTSTAT totalBytesDownloadedPerSession : Integer
    {
      INIT{totalBytesDownloadedPerSession.setValue(new Integer(0));}

      EVENTS{BytesDownloadedCountPerDownload_EventComplete}

      UPDATE{totalBytesDownloadedPerSession.setValue(
                                  totalBytesDownloadedPerSession.getValue() +
                                    BytesDownloadedCountPerDownload_FinalValue);}
    }
  }
}
```

Both the loginCount and downloadCount statistics are analogous in nature, whereby both count the occurrence of an event and are hence declared through the instantiation of definition EventCount. Evaluating the largest byte download count from all sessions requires the specification of a three-layer statistic declaration. Statistic BytesDownloadedCountPerDownload evaluates (for each session) the number of bytes downloaded for each download request, statistic totalBytesDownloadedPerSession evaluates the total number of bytes downloaded per session (by listening to the former statistic's statistical event upon interval closing), whereas statistic largestDownloadCount operates at the global level, receiving the download counts for all sessions (through the statistical event of totalBytesDownloadedPerSession), and keeps the largest received value.

10.4 The Download Penalty Scenario

Bandwidth is an expensive network resource. Nevertheless, due to the lack of a mechanism specifying a fair distribution of bandwidth among clients, situations may arise where certain few use up most of the bandwidth, leaving the many with a disproportionately small share of bandwidth. Hence, we propose a mechanism aiming to induce better fairness with regards to bandwidth distribution. This technique is based on the concept of penalty, whereby each client is given a penalty (in the form of a time delay) based on the amount of client downloads within the last 90 seconds. This penalty is consequently executed on the next requested client download. Hence, such a mechanism penalises clients which frequently download files, thus forcing them to slow down their rate of downloads which in turn frees up bandwidth for other more considerate clients.

```
GLOBAL  {
  FOREACH (Session s)  {
    VARIABLES  {
      int delay = 0;
    }

    EVENTS  {
      downloadStarting() = {ftpdProtocol p.eventDownloadFilePre(arg1)}
                                                          where {s = p.getSession();}
      downloadComplete() = {ftpdProtocol p.eventDownloadFilePost(arg1,arg2)}
                                                          where {s = p.getSession();}
    }

    INTERVALSTAT DCLast90Seconds : Integer  {
      INIT{DCLast90Seconds.setValue(new Integer(0));}

      EVENTS{downloadComplete}

      LIST{true}

      INTERVAL  {  TIME[90 \ ]  }
```

```
      UPDATE[ DCLast90Seconds.getList().add(new Integer(1)); ]
            { DCLast90Seconds.setValue(DCLast90Seconds.getList().size());
              (DCLast90Seconds.getValue() == 0) ? delay = 0; :
              (DCLast90Seconds.getValue() == 1) ? delay = 1000; :
              (DCLast90Seconds.getValue() == 2) ? delay = 2000; :
              (DCLast90Seconds.getValue() >= 3) ? delay = 5000;    }
    }

    PROPERTY executeDelay
    {
      STATES {  BAD { } NORMAL { }  STARTING { start }  }

      TRANSITIONS {
        start -> start[downloadStarting()\ (delay > 0) \
                              try { Thread.sleep(delay);} catch (Exception e){}  ]
      }
    }
  }
}
```

Interval statistic DCLast30Seconds counts the number of downloads within the last 90 seconds through the use of the associated list. Each time a download is complete an element is added to the list. Hence, given that this list is implicitly updated, removing items added to the list more than 90 seconds ago, the list size signifies the actual number of downloads within the last 90 seconds. The statistic update, apart from setting the statistic valuation to the list size, also sets an internal variable delay marking the amount of milliseconds acting as the penalisation amount. Property executeDelay simply executes this penalisation amount by forcing the currently executing thread to sleep the amount specified by the delay variable prior to the start of the next download.

Although this proposal is a simple yet effective means for better bandwidth distribution, it is not without its flaws. For example, the mechanism gives client penalties regardless of whether the system is under stress. Moreover, the mechanism does not factor in the download file size. Hence, alternate techniques have been proposed, such as evaluating the client penalty based on the comparison of the byte count downloaded by the client with the average client byte count, as expressed below

```
STATDEFINITIONS{  PointStatPackage.txt;  }
GLOBAL
{
  EVENTS  {
    sendInfo(OutputStream out) = {call out.write(arg1,arg2,arg3) }
    passwordEvent(String res) = {execution ftpdProtocol p.PASS(arg1) uponReturning (res) }
    downloadStarting(ftpdProtocol p) = {call p.eventDownloadFilePre(arg1)}
    downloadComplete(ftpdProtocol p) = {call p.eventDownloadFilePost(arg1,arg2)}
  }

  POINTSTAT totalBytesDownloaded : Integer {
```

```
    INIT{totalBytesDownloaded.setValue(new Integer(0));}

    EVENTS{BytesDownloadedCountPerDownload_EventComplete()}

    UPDATE{totalBytesDownloaded.setValue(totalBytesDownloaded.getValue()
                                        + BytesDownloadedCountPerDownload_FinalValue);}
  }

  POINTSTAT totalClientsLoggedIn = PointStatPackage.EventCount[passwordEvent \
                                                    (res.equals("230 logged in"))]

  POINTSTAT averageBytesDownloaded : Integer  {
    INIT{averageBytesDownloaded.setValue(new Integer(0));}

    EVENTS{totalBytesDownloaded_Event}

    UPDATE{averageBytesDownloaded.setValue(totalBytesDownloaded_Value /
                                          totalClientsLoggedIn.getValue());}
  }

  FOREACH (Session s)  {
    INTERVALSTAT BytesDownloadedCountPerDownload : Integer  {
      ...
    }

    POINTSTAT totalBytesDownloadedPerSession : Integer  {
      ...
    }

    PROPERTY executeDelay
    {
      STATES  {  BAD { }  NORMAL { }  STARTING { start }  }
      TRANSITIONS  {
        start -> start[downloadStarting() \ (s == p.getSession()) \
                       if (totalBytesDownloadedPerSession.getValue() >
                                        averageBytesDownloaded.getValue())
                       try { int penalty = (totalBytesDownloadedPerSession.getValue()
                                   - averageBytesDownloadedPer.getValue())/1024;
                             Thread.sleep(penalty);}
                       catch (Exception e){}]
      }
    }
  }
}
```

Note that statistics BytesDownloadedCountPerDownload and totalBytesDownloadedPerSession have been purposefully left out since they have already been implemented in the previous section. Statistics collection occurs at two context levels; globally and at a per session basis. Moreover, both levels are linked through the use of statistical events and statistic objects. At the session context level, statistics BytesDownloadedCountPerDownload and totalBytesDown-

loadedPerSession are used to evaluate the total byte count downloaded within that session. Globally, totalBytesDownloaded collects the total number of bytes downloaded by all sessions. This is achieved by listening to BytesDownloadedCountPerDownload's statistical event (upon interval closing), whereby all sessions send the number of bytes downloaded for each of their downloads. Consequently, using statistics totalClientsLoggedIn (instantiating statistic definition EventCount previously defined) and averageBytesDownloaded the system evaluates the average number of bytes downloaded by the system. In truth, statistics averageBytesDownloaded, totalBytesDownloaded and BytesDownloadedCountPerDownload form a three-layer statistic definition, partly across contexts and partly within context. All that is left is to evaluate and execute the download penalty, which is executed by property executeDelay, which simply executes a delay whose size is dependent on the difference between the session and global average behaviour (with respect to byte download counts).

This solution is certainly more flexible and considerate than the previous rigid approach. Not only is the file size factored in, but penalisation is dictated by comparing client behaviour with the average behaviour. Hence, clients which are using most bandwidth compared to their peers are penalised most. Nevertheless, In truth this attempted solution is also not without its flaws, the most concerning being that it is vulnerable to attack by 'teaching' the mechanism. This attack can be executed through concerted client action of heavy downloading, thus gradually raising the average download byte count and hence 'teaching' the system that more downloads should be allowed.

10.5 The User Risk Factor Scenario

Monitors executing on top of the underlying system consume computational resources, and is made worse the more users are logged into the system. In order to alleviate this problem, certain logged in users can be ignored from monitoring after gaining enough of the system's trust of being non-malicious. Although multiple approaches can be taken for the definition of users gaining the system's trust, we shall be taking a simplistic approach defined below. In truth, this scenario offers a basic introduction to the notion of probabilistic monitoring, and shall be expanded by a more comprehensive technique in the following chapter.

```
GLOBAL
{
  FOREACH (Session s)  {
    VARIABLES  {
      Boolean checkProperties = true;
    }

    EVENTS  {
      deleteFileEvent(ftpdProtocol p) = {p.DELE(arg1)} where {s = p.getSession();}
      downloadFileEvent(ftpdProtocol p) = {p.RETR(arg1)} where {s = p.getSession();}
      uploadFileEvent(ftpdProtocol p) = {p.STOR(arg1)} where {s = p.getSession();}
      removeDirectoryEvent(ftpdProtocol p) = {p.RMD(arg1)} where {s = p.getSession();}
      makeDirectoryEvent(ftpdProtocol p) = {p.MKD(arg1)} where {s = p.getSession();}
      ListEvent(ftpdProtocol p) = {p.LIST(arg1)} where {s = p.getSession();}
```

```
  passwordEvent(ftpdProtocol p) = {p.PASS(arg1)} where {s = p.getSession();}
  userEvent(ftpdProtocol p) = {p.USER(arg1)} where {s = p.getSession();}
  privilegedActions(ftpdProtocol p) = { deleteFileEvent(p) | downloadFileEvent(p) |
                                        uploadFileEvent(p) | removeDirectoryEvent(p) |
                                        makeDirectoryEvent(p) | ListEvent(p)  }
  notPasswordEvents(ftpdProtocol p) = { ... }
}

INTERVALSTAT countLastThirtyMinutes : Integer  {
  INIT{countLastThirtyMinutes.setValue(new Integer(0))}

  EVENTS{privilegedActions()}

  LIST{false}

  INTERVAL {
     OPEN[passwordEvent() \ \ countLastThirtyMinutes.setValue(new Integer(0));  ]
     DURATION[1800 \ if ((Integer)countLastThirtyMinutes.getValue() < 30)
                                                    checkProperties = false; ]
  }

  UPDATE{countLastThirtyMinutes.setValue(countLastThirtyMinutes.getValue() + 1);}
}

PROPERTY userPass {
  STATES  {  BAD {sequenceNotAdheredTo }  NORMAL { sequenceMiddle }
                             STARTING { sequenceStart }  }
      TRANSITIONS  {
    sequenceStart -> sequenceMiddle[userEvent()\ checkProperties == true \]
    sequenceMiddle -> sequenceStart[passwordEvent()\ \]
    sequenceMiddle -> sequenceNotAdheredTo [notPasswordEvents\ \p.QUIT(""); ]
  }
}

PROPERTY deleteFilePermission {
  STATES  {  BAD { noPermission }  NORMAL {}  STARTING { legalStatus }  }
  TRANSITIONS  {
    legalStatus -> legalStatus [deleteFileEvent\ ((checkProperties == true) &&
                                                  ftpdPermissions.permissionWrite(
                                                   s.getIdentity())) \  ]
    legalStatus -> legalStatus [deleteFileEvent\ ((checkProperties == true) &&
                                                  ftpdPermissions.permissionWrite(
                                                   s.getIdentity()) == false ) \
                                                   p.QUIT("");  ]
  }
}

PROPERTY listDirectoryPermission {
  STATES  {  BAD { noPermission }  NORMAL {}  STARTING { legalStatus }  }
  TRANSITIONS  {
```

```
        legalStatus -> legalStatus [deleteFileEvent\ ((checkProperties == true) &&
                                                      ftpdPermissions.permissionRead(
                                                       s.getIdentity())) \  ]
        legalStatus -> legalStatus [deleteFileEvent\ ((checkProperties == true) &&
                                                     (ftpdPermissions.permissionRead(
                                                      s.getIdentity()) == false )) \
                                                      p.QUIT("");  ]
      }
    }
  }
}
```

Properties userPass, deleteFilePermission and listDirectoryPermission represent three properties checked for each session, and represent logic which must be adhered as specified by the RFC 959 protocol and the server logic (with respect to permissions. Note that initially all declared properties monitor the session execution. In truth, a real case deployment of such a scenario would define a substantially larger amount of properties. Note that the all property transitions can be turned off using declared internal variable checkProperties. Hence, if the developed mechanism decrees that the current session need not be monitored further, all that is required is to set variable checkProperties to false. The mechanism itself works by defining a set of FTP commands which are considered risky, and counting the occurrence of such events within the first thirty minutes since login. This is implemented by duration interval statistic countLastThirtyMinutes, which counts the number of number of occurrences of any events declared in event collection privilegedActions for the first thirty minutes since login. Upon closing the interval, a choice is made whether to continue monitoring the session. If the number of risky commands executed by the user is above a certain threshold (arbitrarily chosen to be 30), the properties continue monitoring the session. However, if the count is less than the threshold the user is considered to have gained enough trust, and hence all properties are switched off by setting checkProperties to false.

In truth this chosen technique is too trivial and can be easily exploited by simply not executing risky commands for the first thirty minutes. Nevertheless, this example offers an interesting introduction to probabilistic monitoring through the use of our statistical framework, and shall be expanded on below by developing a more complex mechanism which takes into account more parameters.

10.6 Conclusions

The presented case studies are a valid introductory motivation for the practical use of LarvaStat, as well as presenting simplified techniques used in areas such as probabilistic monitoring and intrusion detection (more in the next chapter). In most cases, complex functionality expressed through the use of statistics collection at runtime was presented in a straightforward fashion through the description of a concise LarvaStat script, which is encouraging. We theorise that implementing such functionality directly in the code would involve a considerably larger amount of code and effort. We consider the presented case studies as the practical justification

of the motivated features within our framework, whereby the use of features such as the statistical event as well as the considerable use of all statistical constructs for the expression of real life requirements to be satisfactory. Also of note is the ease with which multilevel statistics, both within the same context and also across contexts, are intuitively implemented. Perhaps the most non trivial construct specification is that of the interval with an associated context. Nevertheless, we consider the solution of overlapping intervals through context to be the best approach, which implies that certain complexity is unavoidable.

11. Intrusion Detection

Chapter 4 introduced various fields which benefit from the development of an online runtime verification framework with statistical capabilities, one of which is intrusion detection. Consequently, this chapter is focused on applying our statistical framework to intrusion detection, with the aim of providing a comprehensive real life scenario depicting the application of our general statistical framework to a field of considerable complexity implying heavy statistical computation. The following chapter involves an overview discussion of intrusion detection, which is followed by a more in depth analysis of anomaly detection (a candidate approach to intrusion detection), since it is this approach which focuses on the use of statistics for intrusion detection. It is within this section that certain issues highlighting the difficulty of statistical intrusion detection are brought to the fore. Following this, an interesting approach to the integration of runtime verification with intrusion detection is presented. Finally, a considerably substantial case study is presented applying multiple statistical techniques to intrusion detection. In truth, this case study goes further, by also applying the notions of system profiling as well as probabilistic monitoring. Note that this case study uses the same ftpd server implementation introduced in the previous chapter.

11.1 Intrusion Detection Overview

Event with the best of effort made in creating security systems capable of withstanding the heaviest of attacks, intrusions still occur. Moreover, whereby the act of intrusion previously required specific skills only few were capable of, nowadays attacks are concerted, fully automated and varied. This points to the need for a mechanism capable of intrusion detection, whereby intrusion detection can be concisely considered to be the field which studies the detection of unauthorised actions which attempt to manipulate or in any way compromise system resources. Multiple approaches have been presented for the implementation of intrusion detection, and can be broadly classified under three approaches:

- **Anomaly Detection** — Based on the assumption that what is abnormal is probably suspicious [4]. Hence, implicit is the belief within anomaly based techniques is that user behaviour follows a statistically predictable pattern [7]. *If a user typically uses document processing software and web applications, it is highly suspicious if the same user suddenly*

starts accessing sensitive system resources.. Concisely, the development of anomaly detection techniques involve designing a mechanism which builds a model representing typical user behaviour, and comparing the current user behaviour with typical behaviour. If large discrepancies are marked, an alarm is sounded.

- **Misuse Detection** — Also sometimes referred to as rule-based detection, misuse detection checks if the current sequence of instructions executed by the user is known to be harmful to the system. *If a sequence of instructions are known to circumvent the system firewall through a loophole in the system, and the user attempts to execute this exact sequence of instructions, then the user is most probably attempting to intrude.* Such harmful instruction sequences are usually encoded within rule sets, whereby the current user behaviour is compared to rules within this set. Although misuse detection techniques are usually very accurate (implying a minimal false alarm rate — more below), they can only detect known attacks and are hence vulnerable to unknown attacks or event variations of known attacks. Hence, the rule sets of such techniques need to be frequently updated if they are to be effective.

- **Specification Based Detection** — Whereas misuse detection takes a default accept approach — only disallowing known harmful behaviour thus allowing for unknown behaviour — specification based detection takes a default deny approach. Hence, techniques for specification detection specify what each application is allowed to do, and any user requiring abnormal application behaviour is marked as harmful to the system. In truth, specification based detection attempts to solve the main issue with misuse detection of being unable to handle unknown attacks. Axelsson [3] categorises specification based detection as being close to anomaly detection, since both approaches attempt to specify what is normal behaviour, and reject abnormalities from what is normal.

There exists no best approach, with most intrusion detection systems using various techniques from all approaches for best results, as discussed in [3]. In fact, all three approaches imply their own particular advantages and disadvantages. Whereas anomaly detection is capable of recognising previously unknown attacks through the characterisation of abnormal behaviour, it is (perhaps unavoidably) prone to unacceptable false alarm rates, as well as being ineffective due to missing certain attacks masked as normal behaviour (by 'teaching' the intrusion detection system). Moreover, unusual behaviour is not necessarily harmful behaviour, which increases further the risk of false alarm rates. On the other hand, misuse detection is very accurate, capturing most (if not all) known attacks. Nevertheless, the nature this approach implies that it will forever be prone to being massively ineffective with respect to unknown attacks. Hence, misuse detection in an environment with rapidly evolving, novel attacks, most of which are not fully characterised is useless. Specification based detection directs its attention to this problem, and should theoretically perform at a much higher effectiveness rate than misuse detection on unknown attacks. Nevertheless, as specified by Bishop [7] specification based detection is still in its infancy and requires research, not to mention the massive effort required in specifying normal behaviour for all known software.

Although Axelsson [3] mentions multiple issues with intrusion detection systems, perhaps the most crucial is that of effectiveness. Effectiveness refers to the success rate when identifying

intrusion attacks, as well as the rate at which it marks non-harmful behaviour as intrusive. These rates are frequently referred to as the detection and false alarm rates, and shall be discussed in further detail below.

11.2 Anomaly Detection In Detail

Although intrusion detection admits multiple techniques characterised under three broad categorisations, we shall be focusing on anomaly detection since it is the only approach which focuses on the use of statistics. As previously stated, anomaly detection is all about characterising normal (and abnormal) behaviour, such that if current behaviour varies sufficiently from the normal behaviour it is considered suspicious. Certain issues intrinsic with anomaly detection stand out, the first being the implicit need for a certain stage of observation and learning what makes abnormal behaviour. Another observation implies that apart from the mechanism for characterisation of normal behaviour, there is the requirement of a decision making mechanism, determining the degree of variance from normal behaviour for it to be deemed anomalous. As presented by Axelsson [4], techniques applied to anomaly detection can be broadly classified under two classifications, these being self-learning and programmed techniques which shall be discussed below.

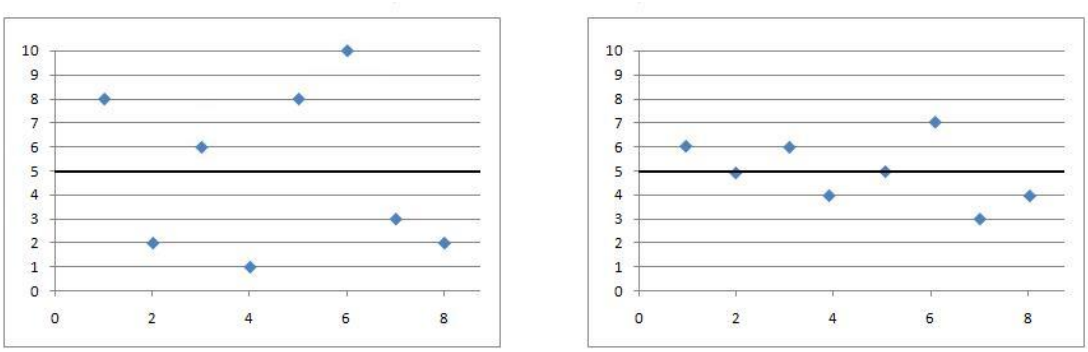
Self-learning techniques refer to techniques which learn what constitutes normal behaviour by observing actual behaviour during system execution. Axelsson [4] further dissects self-learning techniques into time based and non-time based. In other words, non time based techniques model normal user behaviour through stationary processes [21] without factoring the change in behaviour as time progresses. Such techniques include the use of statistical moments (more below)and rule modeling systems, whereby rules are induced which characterise normal behaviour. On the other hand, time based techniques factor in the notion of behaviour changing with time, and use mechanisms such as artificial neural nets and markov chains.

Axelsson [4] defines programmed techniques as techniques which require someone, usually the user or system developer, to 'teach' the system what constitutes normal behaviour. Such programmed techniques usually involve the collection of a predefined set of simple statistics, possibly defining a set of rules (thus hinting at specification based detection) or even defining a set of pre-programmed thresholds stating when to raise the alarm.

Three core mechanisms (briefly mentioned above) have been presented by Denning [16] for the implementation of anomaly detection using statistical techniques which serve as the cornerstone for most subsequent techniques attempting statistical anomaly detection. Although already briefly mentioned in the above anomaly detection taxonomy, we shall discuss each mechanism in more detail below (for more information see also [7]).

The first statistical technique identified by Denning is the threshold metric, whereby an event occurs a number of times bound by a minimum and maximum threshold. An example of such a threshold technique is that of raising an alarm if more than three consecutive bad logins occur (thus the first case study in section 10.2 defining an action after three consecutive

bad logins can be considered to be a simple case of anomaly detection using a programmed threshold technique). The second technique involves the use of statistical moments, whereby the system collects valuations for such moments (usually the average and variance) which jointly statistically depict normal user behaviour. Whereas the use of the average statistic is evident (since the average can characterise a data set distribution, thus directly representing what is 'normal'), the use of variance is perhaps less evident.



Variance is considered to be a measure of the statistical dispersion in a data set, and is calculated by averaging the distance squared from the average value [34]. In other words, variance is a measure of how much the data set is spread out. Hence, although the data sets plotted above both admit the same average value, the first data set admits a considerably larger variance than the second since the data points are considerably more spread out from the average. Returning to the issue of anomaly detection, the measure of variance of a particular data set characterising user behaviour offers valid insight, since the variance acts as a measure of predicability within a data set. This implies that anomalous behaviour by a user who normally admits an acceptable behaviour with a low variance is far more suspicious than anomalous behaviour coming from a user whose behaviour varies wildly (hence having a large variance). The last technique presented by Denning is the use of markov models, whereby user behaviour is conceptualised as a sequence of states, and the transition from a state to the next is governed by probabilities (either pre-programmed or learnt from the system execution) as explained in [7]. Hence, given a user who traverses a low probability transition (or possibly a chain of low probability transitions) is deemed to be anomalous in behaviour. Note that all three techniques proposed by Denning shall be applied in conjunction for the development of the intrusion detection system defined for the case study below.

11.3 The Difficulty with Intrusion Detection

Although there is a great need for intrusion detection systems which are to satisfy multiple requirements, the most important of which is that of effectiveness, the nature of the field admits certain damning issues which implies that intrusion detection is more difficult than first thought [3]. The following section looks at the work presented by Axelsson [3] and Helman et al. [21] discussing the difficulties with intrusion detection. Whereas Axelsson introduces the issue of the

base rate fallacy as applied to intrusion detection, Helman presents a theoretical framework for the characterisation of intrusion detection, and gradually introduces this framework to real life situations such as incomplete information and the vast number of events monitored by the typical system. Recall that effectiveness for intrusion detection systems is defined as the combination of the detection rate (how many of the overall attempted intrusions are caught) and the false alarm rate (how many raised alarms proved to be false). As we shall see below, both works cast doubt as to whether desirable effectiveness levels for intrusion detection systems are attainable.

The Base Rate Fallacy

The base rate fallacy is a fundamental notion which lies at the core of bayesian statistics, and is best characterised by an example which shall be presented below[1]. Recall that bayes' theorem is defined as

$$P(A \mid B) = \frac{P(A)\,P(B \mid A)}{\sum_{i=1}^{n} P(A_i)\,P(B \mid A_i)}$$

The following example captures the essence of the base rate fallacy. Suppose the availability of an oracle capable of predicting the future with 98% accuracy. Hence, 98 of the 100 responses to questions posed to the oracle come to pass. Suppose we buy a lottery ticket, of which there are one million, and ask the oracle whether we shall win the lottery with this ticket. Given that the oracle (incredibly enough) responds that we shall win the lottery, what is the actual probability of winning given the above setting?

Let us evaluate the probability using Bayes' theorem, whereby P denotes a positive result by the oracle (hence signifying that the oracle believes that we shall win the lottery) and W denotes the probability of winning the lottery. Given that by definition only one lottery ticket can win from the million sold, this implies that P(W) = 1 / 1000000. Also, assuming that we do actually win, the probability of the oracle predicting the win is 98%, hence P(P|W) = 0.98 (analogously P($\neg$P| $\neg$W) = 0.98). Given the above setting, the probability we require is P(W|P) i.e. the probability of winning given that the oracle response. Using bayes' theorem (above), this probability is encoded as follows

$$P(W \mid P) = \frac{P(W)\,P(P \mid W)}{P(W)\,P(P \mid W) + P(\neg W)\,P(P \mid \neg W)}$$

The only value not immediately known is $P(P \mid \neg W)$, which using standard probability identities ($P(A \mid \neg B) = 1 - P(\neg A \mid \neg B)$) is equal to 0.01. Hence, by plugging in all these values into the above equation $P(W \mid P) = 0.00009799$, which unfortunately implies that we will still probably not be winning the lottery. This result is insightful, since even with a very accurate oracle (98% accuracy) the low rate of incidence (i.e. the odds of actually winning), also known as the base rate, 'overrules' the rate of prediction. In other words, an accurate predictor of a rare event is still fruitless when faced with the incredible rarity of the event occurrence.

[1]Note that the presented example is analogous to the example given in [3].

As presented by Axelsson, the base rate fallacy is directly applicable to the problem of intrusion detection, whereby the oracle is replaced by the intrusion detection system and winning the lottery is replaced by detecting an intrusion. Typically, an intrusion detection system monitors events in the order of millions of which not more than 100 are malicious. Hence, keeping this typical setting in mind, the following summarises the application of the best rate fallacy to intrusion detection (taken from [3]).

Suppose I denotes intrusion, and A denotes alarm raised. Hence, P(I) denotes the probability of an intrusion, P(A|I) denotes the detection rate and P(A| ¬I) denotes the false alarm rate. What we require for an effective intrusion detection system is the maximisation of P(I|A) i.e. whenever an alarm is sounded, there is a high probability of an intrusion occurring. Using bayes' theorem this is encoded as

$$P(\,I \mid A\,) \;=\; \frac{P(\,I\,)\,P(\,A \mid I\,)}{P(\,I\,)\,P(\,A \mid I\,) \;+\; P(\,\neg I\,)\,P(\,A \mid \neg I\,)}$$

Given that we are dealing with probabilities, all values plugged into the equation are fractions ≤ 1. Moreover, since the numerator involves the multiplication of two fractions, the result is also ≤ 1. Given that we want to maximise P(I|A), we hence require the minimisation of the above denominator. Now, since the probability of intrusion is very low (one in a million), this conversely implies that the probability of no intrusion is very high. Consequently the denominator valuation is governed by the false alarm rate ($P(\,A \mid \neg I\,)$) since it is this term which is multiplied by $P(\,\neg I\,)$, whereas the other term is multiplied by $P(\,I\,)$, rendering it negligible. In conclusion, this informal reasoning implies that the construction of an effective intrusion detection system is bound by its false alarm rate. In other words, we must keep the false alarm rate (perhaps unattainably) low. Axelsson [3] goes further by analysing the above equation in realistic settings, which sheds further light on the difficulty of intrusion detection. In fact, given a realistic detection rate of 0.7, having a 0.001 false alarm rate reduces P(I|A) to approximately 0.02, which is unacceptable. Hence, if an intrusion detection system is to be effective it must have an almost negligible false alarm rate, which is currently unattainable for most currently available techniques in intrusion detection. This problem is made worse for anomaly detection techniques, since such techniques identify anomalies rather than intrusions (hoping for a correlation between both), which implies a substantially higher false alarm rate.

A Theoretical Framework

Helman et al. [21] presents a theoretical framework representing intrusion detection techniques, as well as introducing theoretical bounds on performance and effectiveness given multiple settings. A typical process is modeled using three stochastic stationary processes as defined below (definition and explanation taken from [21])

$$H(\,t\,) = \;\begin{cases} N(\,t\,) & \text{if } D(\,t\,) \;=\; 1 \\ M(\,t\,) & \text{if } D(\,t\,) \;=\; 0 \end{cases}$$

Given a system process represented by H, the t^{th} transaction (event) in the audit trail (event sequence) is either a normal or malicious activity, as determined by stochastic process D. Hence,

if D(t) = 0, the process acts normally as specified by stationary stochastic process N, whereas if D(t) = 1 the process acts maliciously as specified by stationary stochastic process M. Note that by definition the representation of processes using stationary processes implies that the above definition assumes that the process behaviour does not alter with time. Hence, statistical values such as the average and variance remain constant.

Given this setting, Helman (also basing his work on bayes theorem) analyses the theoretical bounds on intrusion detection system effectiveness under different circumstances. These circumstances include a formal analysis with perfect information (hence with all three stationary processes fully characterised), and two distinct analysis with imperfect information. Such imperfect information includes the lack of characterisation of process D, and consequently the formal analysis of intrusion detection effectiveness with incomplete characterisation of processes N and M. Without going into the details of the analysis, Helman shows the difficulty induced with imperfect information especially when faced with the vast number of events which require monitoring. Moreover, attempting to reduce the number of events through techniques such as attribute selection and value aggregation is proven to be NP-hard. These results are indeed unfortunate, since real life scenarios usually imply imperfect imperfect information and lengthy event traces. Helman concludes by also discussing other non-modeling approaches including the use of clustering, heuristics and rule based detectors.

11.4 Related Work

The work presented by Sekar et al. [35] offers a valid basis which introduces the application of runtime verification to intrusion detection. Moreover, this work also introduces the requirement of ftpd server monitoring and also gives us valid insight into the specification of required properties.

Without directly referring to the field of runtime verification, Sekar presents a domain specific runtime verification framework which allows for the definition of a rule based intrusion detection system, whose aim is to produce fast monitors inducing minimal overhead on the system. Rules are expressed through a defined logic (also capable of reparation through the execution of defined function calls), which are consequently converted into an online monitor automatically instrumented with the system under consideration. Sekar presents regular expressions for events (REE), which is the defined logic allowing for the expression of the required rules and essentially extend regular expression by allowing the use of variables. A typical REE exprssion applied on an ftpd server implementation is the following (whereby c denotes the client connection)

```
begin(c);(!authenticate(c))*;open(/etc/passwd,c) -> terminate(c)
```

which specifies that no client can attempt to access the password file prior to authentication. Hence, if such action is attempted the client connection is immediately terminated. Note the use of variable c throughout the expression, whereby the same client connection is passed between distinct system events. REEs are converted to extended finite state automata (EFSA) for their execution, whereby analogously to REEs extending regular expressions, extended finite state automata augment finite state automata (FSA) by allowing for automata to assign

or examine a finite set of variables. EFSAs come in two flavours (also analogously to FSAs), these being the non-deterministic extended finite state automata (NEFA) and deterministic finite state automata (DEFA). Using an analogous approach to the conversion from REs to FSAs, a straightforward conversion between REEs and EFSAs exists. However, converting REEs to a NEFA returns an automaton which is inefficiently simulated, whereas the conversion from REEs to a DEFA returns an automaton which although efficiently simulated, is space inefficient.

Given this conundrum, Sekar presents a novel algorithm detailing an average case solution to this problem of inefficiency. This resulting algorithm converts REEs to an EFSA which finds a balance between the space explosion and runtime inefficiency imposed by the DEFA and NEFA respectively, and is geared towards avoiding the size explosion implied by state variables [35]. Hence, the result is an algorithm which produces an automaton exponential in size in the worst case, however in practice rarely requires more than a reasonable amount of time execution and space. Moreover, this algorithm is proven in [35] to produce a DEFA for REE(0) rules (a subset of rules expressible by REEs, whose event argument values cannot be referred to except immediately after the event occurrence).

Concisely, the presented algorithm integrates all specified rules into one EFSA whose size eventually converges to an approximate value as more rules are added. The result is an automaton approximately insensitive to the complexity and number of specified rules [35], whereby adding new rules does not necessarily increase the automaton size. Moreover, in practice the overhead induced on the underlying system is within the 1 - 2% range, which makes this technique very fast.

Comparing this approach to our work, it is apparent that REEs are not as expressive as LarvaStat, since REEs are not sufficiently expressive for the specification of real time or statistical properties. Moreover, REEs are domain specific bound to the expression of intrusion detection systems, whereas LarvaStat is a more general approach whose potential capability is beyond intrusion detection. Nevertheless, the focal point of this approach is speed and the reduction of overhead on the system, which this approach is very good at. Although the overhead induced by LarvaStat is yet to be properly analysed, the results presented for LARVA (which LarvaStat is converted to) imply that the approach presented by Sekar [35] outperforms LARVA in this respect.

11.5 Case Study

Computational resources are expensive, and need to conserved as much as possible. However, monitoring the behaviour of all users is expensive, especially when using complex statistical measures for the characterisation of their behaviour. This implies a situation where the requirements of system security conflicts with the requirement of efficiency and fast response times. We hence devise a mechanism which reaches a compromise between both requirements through the notion of probabilistic monitoring, which shall be implemented below. Using the same ftpd server implementation discussed in the previous chapter as the underlying system, we shall build an intrusion detection system which is probabilistic in nature. Hence, although all new users are

initially monitored, if a user gains enough of the system's trust through consistently acceptable behaviour, and the system is under a sufficiently heavy load, we may opt not to monitor such users. This implies that user monitoring is executed only if the user in question is deemed sufficiently risky, and/or the system is not under excessive stress. The development of this system entails the implementation of two distinct yet interdependent components:

- **A System Profiler** — A component responsible for quantifying the system load factoring in the current load as well as the system load history.
- **An intrusion detection system** — Using statistical anomaly detection techniques previously discussed, as well as misuse detection techniques in the form of rule specifications, we implement an intrusion detection system which monitors user behaviour in search for anomalous or illegal behaviour, and if such behaviour is encountered appropriate action is taken.

Given the information collected by both components, each time a user logs in the system has to make a decision whether to monitor the user or not. This decision is taken by factoring in the system load and user risk factor as shall be discussed below. An issue with making such decisions upon login is the fact that we require information regarding user behaviour which is not available at runtime (since the user just logged in). We therefore require a mechanism which keeps track of the user behaviour history even between user logins. This issue is also applicable to the system load history which we require to implement within the system profiler. This information persistence mechanism is implemented using an SQL database, storing (i)the system load history (ii) user behaviour history (iii) a probabilistic check log. This SQL has been implemented using a MySQL database and implemented within the intrusion detection system, hence requiring no alteration to the original system.

Given the complex nature of the system, it would be unreasonable to give its implementation all at one go. We shall hence discuss the implementation at a per component bases in an evolutionary manner, finally integrating both components during the discussion regarding the probabilistic monitoring choice mechanism. For the complete script see appendix C.

System Profiler

The following system profiler is a heuristic based profiler which analyses the system's current load as well as the load history, and quantifies this information into a value v, $0 \leq v \leq 1$ representing what the profiler believes to be the system load at that instant in time. The system profiler bases its evaluation on two assumptions, the first being that the ftpd server's expensive resource is its bandwidth, while the second being that the ftpd system's history is an indication of its load in the future.

Based on the first assumption, the current system load is evaluated through the inspection of the current bandwidth usage. Note that bandwidth is essentially consumed by the current open download and upload connections. Also note that the system profiler adopts the following heuristics (i) if the number of open connections is less than 20 the system is under no significant load (ii) Each user should not on average have more than 5 open connections. (iii) The

current load is based on the average number of open user connections, whereby each user having on average more than 5 open connections automatically implies that the system is under a heavy load. Obviously such heuristics are only approximate in their description of the system load, and in truth are usually set by experienced professionals such as the system security officer. Nevertheless, these heuristics appear to operate well, and are formulated in the following algorithm.

```
currentTransfers = currentDownloads + currentUploads

if (currentTransfers > 20)
{
  transfersPerUser = currentTransfers / currentLoggedInUsers
  if (transfersPerUser > 5)
    return 1;
  else
    return (currentTransfersPerUser / 5);
}
else
  return 0;
```

Such an algorithm requires the expression of three statistics, namely the number of currently open downloads, uploads as well as the number of currently logged in user and shall be implemented in the script below. Note that the statistic definition eventCount (defined below) shall be henceforth assumed

```
POINTSTAT EventCount : Integer {
  INIT{#EventCount#.setValue(new Integer(0));}
  UPDATE{ #EventCount#.setValue(#EventCount#.getValue() + 1); }
}
```

where eventCount represents point statistics whose nature is to keep count of a particular event. Hence, the LarvaStat implementation of the mechanism evaluating the current system load is defined as (event declarations and method implementations shall be omitted for brevity — see appendix C for full script)

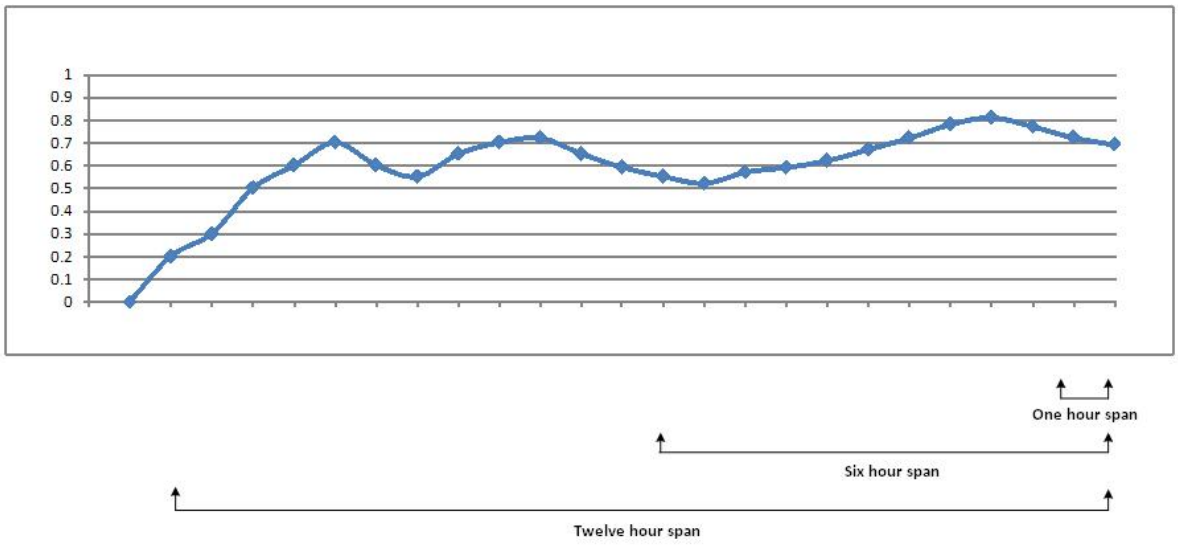
```
STATDEFINITIONS{  PointStatPackage.txt;  }
GLOBAL  {
  VARIABLES  {  Connection connection;  Clock c = new Clock();  }

  EVENTS { ... }

  POINTSTAT DownloadStartingCount = PointStatPackage.EventCount[downloadStarting() \ ]

  POINTSTAT DownloadCompleteCount = PointStatPackage.EventCount[downloadComplete() \ ]

  POINTSTAT CurrentActiveDownloadCount : Integer  {
    INIT{CurrentActiveDownloadCount.setValue(new Integer(0));}
```

```
    EVENTS{downloadEvent}

    UPDATE{ CurrentActiveDownloadCount.setValue(DownloadStartingCount.getValue() -
                                                DownloadCompleteCount.getValue()); }
  }

  POINTSTAT UploadStartingCount = PointStatPackage.EventCount[uploadStarting() \ ]

  POINTSTAT UploadCompleteCount = PointStatPackage.EventCount[uploadComplete() \ ]

  POINTSTAT CurrentActiveUploadCount : Integer  {
    INIT{CurrentActiveUploadCount.setValue(new Integer(0));}

    EVENTS{uploadEvent}

    UPDATE{ CurrentActiveUploadCount.setValue(UploadStartingCount.getValue() -
                                                UploadCompleteCount.getValue()); }
  }

  POINTSTAT CurrentTotalUsersLoggedIn = PointStatPackage.EventCount[passwordEvent \
                                                          (res.equals("230 logged in"))]

  POINTSTAT CurrentTotalUsersLoggedOut = PointStatPackage.EventCount[logout \ ]

  POINTSTAT CurrentUsersLoggedIn : Integer  {
    INIT{CurrentUsersLoggedIn.setValue(new Integer(0));}

    EVENTS{loginLogout}

    UPDATE{ CurrentUsersLoggedIn.setValue(CurrentTotalUsersLoggedIn.getValue() -
                                            CurrentTotalUsersLoggedOut.getValue()); }
  }

  < System Load History Implementation >

  FOREACH (Session s)  { ... }
}
METHODS  {
  calculateCurrentLoad(Integer currentDownloads,Integer currentUploads
                       ,Integer currentLoggedInUsers)  { ... }
}
```

The statistic defining the current number of downloads is evaluated using three point statistic definitions, whereby the first point statistic counts the total number of initiated downloads, the second counts the total number of completed downloads and hence the current download count is the difference between both values. This logic is analogously repeated another two times for the evaluation of the current number of uploads and the current number of logged in users. Method calculateCurrentLoad implements the previously defined algorithm, whereby passing the current valuation of statistics CurrentActiveDownloadCount, CurrentActiveUploadCount

and CurrentUsersLoggedIn as parameters results in the method returning a valuation for the current system load.

The second assumption made by the profiler is that the ftpd server load history is an indication of its future load. In other words this implies that the ftpd server load changes gradually, and very rarely does it sharply change in value. Hence, given a system which is consistently loaded for the past few hours, it will most probably continue under this state for the next few minutes. This reasoning is at the base of the system load history analysis implemented by the profiler as exemplified below



where each point (sampled at a rate of twice an hour) represents the current system load (evaluated using the above algorithm) at that instant. Note how the above graph depicts no sharp change in load, and any overall change is gradual. The profiler evaluates the system load history through the analysis of these current system load samples. Given that an ftpd server (given a typical setting) executes for an extensive time period, the load history is based on the last twelve hours of execution. Moreover, the last hour is more indicative of the future and hence is given more importance. Likewise, the last six hours are more indicative than the last twelve, and are given more importance than the last twelve hour history, although less importance than the last hour's history. Given this logic the average system load history is evaluated for the last hour, last six hours and also within the last twelve hours and a weighted average of all resulting valuations is executed as follows

$$\text{lastHourAverage} \times 0.75 + \text{lastSixHourAverage} \times 0.15 + \text{lastTwelveHourAverage} \times 0.1$$

Whereby the average load history for the last hour is given the highest weighting of 0.75 of the global load history. Note that these (and all succeeding) weights are chosen based on heuristics, and given a real life deployment of this system can be easily altered based on experts' advice.

```
STATDEFINITIONS{  PointStatPackage.txt;  }
GLOBAL  {
  VARIABLES  {  Connection connection;  Clock c = new Clock();  }

  EVENTS { ... }

  < Current System Load Implementation >

  INTERVALSTAT downloadsLast30Minutes : Integer  {
    INIT{downloadsLast30Minutes.setValue(new Integer(0));}

    EVENTS{downloadComplete()}

    LIST{true}

    INTERVAL{  TIME[1800 \  ]         }

    UPDATE[downloadsLast30Minutes.getList().add(new Integer(1));]
          { downloadsLast30Minutes.setValue(downloadsLast30Minutes.getList().size()); }
  }

  INTERVALSTAT uploadsLast30Minutes : Integer  { ... }

  INTERVALSTAT loggedInUsersLast30Minutes : Integer  { ... }

  PROPERTY handleDatabase  {
    STATES  {  BAD { }  NORMAL { }  STARTING { start }  }
    TRANSITIONS  {
      start -> start[serverCore_ObjectCreated()\ \ <Create MySQL Connection>]
      start -> start[DumpToDatabase()\ \ <Dump System load to database based on
                                          traffic observed in last thirty minutes >
                                          c.reset(); ]
    }
  }

  FOREACH (Session s)  { ... }
}
METHODS  {
  calculateCurrentLoad(Integer currentDownloads,Integer currentUploads
                       ,Integer currentLoggedInUsers)  { ... }
  calculateHistorySystemLoad(Connection c) { ... }
}
```

Since the system load for the purpose of evaluating the load history is sampled at a rate of once every thirty minutes, the same algorithm (described above) is evaluated based on the system load within that half an hour between samples. Hence, three additional statistics are implemented named downloadsLast30Minutes, uploadsLast30Minutes and loggedInUsersLast30Minutes. Note that all three statistics are identical in nature, with the alterations being the point of interest declaration. An additional property handleDatabase is also implemented, whose job is to initialise the MySQL connection upon startup, and to periodically dump the

current system load (evaluated using calculateCurrentLoad as applied to the three previously mentioned statistics) to the database. Consequently, method calculateHistorySystemLoad uses this logged information to evaluate the system history load using the approach defined above. Note that ceratin code executing SQL statements have been emitted for brevity. The full code is available in appendix C. Note that in both cases evaluation is executed at a global level, since we require global statistics regarding the overall system performance.

Given the presented implementations for the quantification of the current load and the load history, all that remains is the integration of both values thus representing the overall system load. This is again evaluated using weighted average as described below

$$(\,\text{currentLoad} \times 0.6\,) + (\,\text{loadHistory} \times 0.4\,)$$

which implies that the evaluation of the current load is given slight more importance than the load history. This is implemented through an additional method declaration systemLoad, which applies the above weighting to the valuations of the current system load and the load history, outputting a valuation representing the overall system load.

Intrusion Detection System

The implementation of the intrusion detection system involves the use of multiple techniques previously discussed and is broadly dissected into two components, these being the misuse detector and anomaly detector.

Misuse detection has been defined as checking if the current instruction sequence executed by the user is known to be harmful to the system, and is usually encoded within rule sets. These rule sets shall be expressed in the form of LARVA properties, as defined below

```
GLOBAL  {
  VARIABLES  {  Connection connection;  Clock c = new Clock();  }

  EVENTS { ... }

  < Current System Load Implementation >

  < System Load History Implementation >

  FOREACH (Session s)  {
    VARIABLES {
      boolean userMonitored = true;
      ...
    }

    PROPERTY userPass  {
      STATES {  BAD {sequenceNotAdheredTo } NORMAL { sequenceMiddle }
                STARTING { sequenceStart }  }
      TRANSITIONS  {
```

```
        sequenceStart -> sequenceMiddle[userEvent()\ (userMonitored == true) \]
        sequenceMiddle -> sequenceStart[passwordEventUser()\ \]
        sequenceMiddle -> sequenceNotAdheredTo
                          [ notPasswordEvents \ \ s.close(); ]
    }
  }

  PROPERTY renameSequence  { ... }

  PROPERTY deleteFilePermission  { ... }

  PROPERTY listDirectoryPermission{ ... }
 }
}
```

Only one property definition has been presented for brevity, whereby property userPass specifies that no user can attempt to enter the password before specifying the username. All defined properties are approximately of the same complexity usually specifying properties regarding sequence of events, or if the user has the required permissions for the execution of certain commands. In truth, a real life case scenario would imply the requirement of a substantially greater amount of rules. However, we consider this implementation to be at the proof of concept level, and hence we shall limit ourselves to the property implementations above. Nevertheless, note that a vast number of additional properties can be specified for an ftpd server implementation, as exemplified by [35]. An important observation is that all properties are defined within the session context, which implies that all sessions shall be monitored. Also, unlike anomaly detection techniques (presented below) attempting to quantify user risk, if a property specified for misuse detection is broken, the user connection is immediately terminated due to attempting to execute an illegal action. Finally, note that all properties are governed by the userMonitored boolean which shall be used to switch off properties if the monitor requires so.

The anomaly detection techniques employed in the implemented intrusion detection system implements the three techniques presented by Denning [16], with particular emphasis placed on the use of statistical moments and markov chains. Note that the following methods are implemented

```
calculateNextVarianceValue(currentVariance, currentAverage, n, newValue);
calculateNextAverageValue(currentAverage, n, newValue);
```

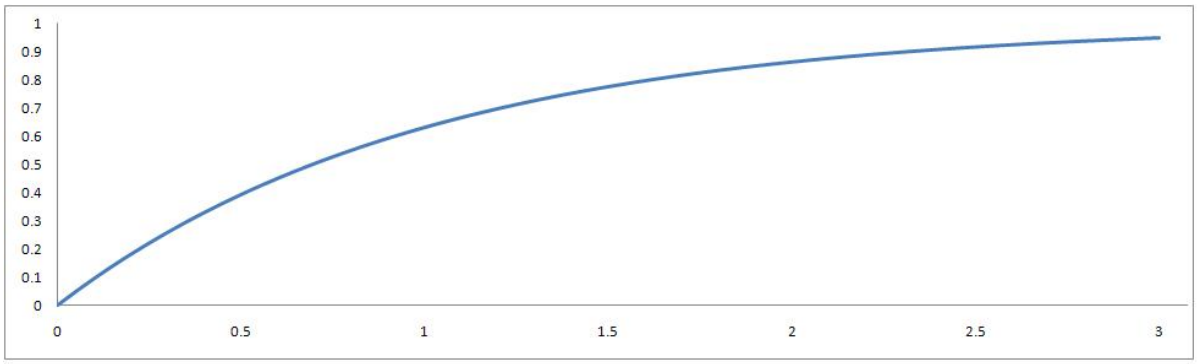
which implement the algorithms presented by Knuth [28] for the incremental computation of the variance and average, given current valuations and the new additional value. It is through these algorithms that complex statistical valuations (especially complex given a long list of values) are efficiently computed.

Since bandwidth is the most precious resource handled by an ftpd server, we shall be monitoring for anomalous user behaviour based on their actions with respect to their management of bandwidth (hence their upload and download behaviour). Given a user who usually (hence implying a low variance) sparingly makes use of bandwidth resources, only to suddenly start

consuming a copious amount of resources (compared to the average) is very suspicious, and may point to an intruder downloading multiple available files on the server. On the other hand, a user who's behaviour does not conform to a statistically predictable pattern (hence implying a large variance) cannot be easily used to determine whether current behaviour is anomalous. This logic is encapsulated in the following algorithm (implemented in the system) responsible for quantifying the user risk factor based on the evaluation of statistical moments.

```
if (noOfLogins > 10)  {
  if (downloadVariance < 20)  {
    if (GlobalDownloadinLast30MinutesAverage < CurrentDownloadInLast30Minutes)  {
      return (-e^(-(CurrentDownloadInLast30Minutes -
                                          OverallDownloadinLast30MinutesAverage)) + 1);
        }
  }
  else
    return 0;

  if (uploadVariance < 20)  {
    if (GlobalUploadinLast30MinutesAverage < CurrentUploadInLast30Minutes)  {
      return (-e^(-(CurrentUploadInLast30Minutes -
                                          OverallUploadinLast30MinutesAverage)) + 1);
    }
  }
  else
    return 0;
}
else
  return 1;
```

Note that GlobalDownloadinLast30MinutesAverageGiven and GlobalUploadinLast30- MinutesAverage refer to the previously defined statistics counting the global number of downloads and uploads occurring within the last 30 minutes. Given that the anomaly detection technique using statistical moments requires a period of learning prior to forming an opinion of what constitutes normal behaviour, the first ten user logins are taken for this purpose of information collection. Hence, any user prior to logging in for 10 times is considered risky to the system. Once enough information is collected user behaviour is analysed for anomalies. If user behaviour is statistically predictable, hence implying a low download variance (empirically chosen to be < 20), user behaviour is compared with the global behaviour. If user behaviour is worse than the global average behaviour, an exponentially weighted value is extracted from the difference between user and average behaviour, giving a value ≥ 0, but ≤ 1 as described below.



whereby a user which downloads or uploads approximately three more than average is considered to have a high risk. The above algorithm implies that given the collection of an additional set of statistics specifying the download count variance, download count average, upload count variance and upload count average we can execute a check for anomalous behaviour.

```
GLOBAL  {
  < Current System Load Implementation >

  < System Load History Implementation >

  FOREACH (Session s)  {
    VARIABLES {
      boolean userMonitored = true;
      Clock userClock = new Clock();
      ...
    }

    POINTSTAT totalLoginCount : Integer  {
      INIT{ totalLoginCount.setValue(new Integer(0));  }

      EVENTS{        userLogin() }

      CONDITION{ result.equals("230 logged in") }

      UPDATE{ totalLoginCount.setValue(totalLoginCount.getValue() + 1); }
    }

    INTERVALSTAT userDownloadsLast30Minutes : Integer  {
      INIT{userDownloadsLast30Minutes.setValue(new Integer(0));}

      EVENTS{userDownloadComplete()}

      LIST{true}

      INTERVAL{  TIME[1800 \  ]  }

      UPDATE[userDownloadsLast30Minutes.getList().add(new Integer(1));]
```

```
            { userDownloadsLast30Minutes.setValue(userDownloadsLast30Minutes.
                                                              getList().size()); }
    }

    POINTSTAT userDownloadsLast30MinutesVariance : Double  {
      INIT{ userDownloadsLast30MinutesVariance.setValue(new Double(0.0));  }

      EVENTS{        userDumpToDatabase }

      UPDATE{ userDownloadsLast30MinutesVariance.setValue(
                                                 calculateNextVarianceValue(...));}
    }

    POINTSTAT userDownloadsLast30MinutesAverage : Double  {
      INIT{ userDownloadsLast30MinutesAverage.setValue(new Double(0.0)); }

      EVENTS{        userDumpToDatabase }

      UPDATE{ userDownloadsLast30MinutesAverage.setValue(
                                               calculateNextAverageValue(...)); }
    }

    INTERVALSTAT userUploadsLast30Minutes : Integer  { ... }

    POINTSTAT userUploadsLast30MinutesVariance : Double  { ... }

    POINTSTAT userUploadsLast30MinutesAverage : Double  { ... }
    ...
    PROPERTY userHandleDatabase  {
      STATES  {  BAD { }  NORMAL { }  STARTING { start }  }
      TRANSITIONS  {
        start -> start[setIdentity \ (userMonitored == true) \
                       <Load Information from Database >  ]
        start -> start[userDumpToDatabase \ (userMonitored == true) \
                       <Dump current information to database > userClock.reset(); ]
      }
    }
  }
}
METHODS{
  calculateNextVarianceValue(...);
  calculateNextAverageValue(...);
  calculateIntrusionProbabilityVariance(...);
}
```

Note that the implementation for the collection of statistical information for uploads is analogous to that defined for downloads. Also note the use of the implemented methods (within the statistical update functions) for the incremental computation of the average and variance valuations. Moreover since the count, average and variance values all trigger on the same event (on the event marking the required sampling of these valuations) the defined statistical ordering

determines the order with which the statistics are updated. The chosen ordering of evaluating the variance prior to the average is required by the online algorithm defined by Knuth, since the computation of the variance depends on the valuation of the average. Property userHandleDatabase is responsible for loading the statistics to their prior latest valuation upon user login, as well as to periodically sample the statistical valuations so as to dump this information to database for later use. Finally, method calculateIntrusionProbabilityVariance implements the above algorithm thus estimating a factor representing the degree of abnormality present in the current user behaviour (using only the above statistics).

The intrusion detection system also implements a markov chain analysing user behaviour as a sequence of events. Each event is assigned a risk factor, and as the user executes commands (thus building a chain of events) an idea regarding the degree of risk involved in the user behaviour is built based on the multiplication of the commands risk factors executed by the user. Note that similarly to the implemented anomaly detection technique using moments (above), this risk factor implied by the markov chain is also sampled and dumped to the database for further use (seen below). Implementation of the markov chain has already been thoroughly discussed in the third case study presented in section 10.2, and will hence not be discussed in detail. The following two construct implementations are required for the simulation of the markov chain

```
POINTSTAT ProbabilityChainStat : Double { ...}

PROPERTY sendRiskProbability { ...}
```

whereby point statistic ProbabilityChainStat keeps track of the chain multiplication of events executed by the user, and property sendRiskProbability captures the execution of events of interest to the markov chain and notifies the ProbabilityChainStat statistic of the risk factor associated with this event execution.

Whereas we have successfully implemented methods for the quantification of anomalous user behaviour based on two anomaly detection techniques, namely statistical moments and markov chains, we are yet to define the action executed by the system in response to these values. This action is defined through the use of pre-programmed thresholds (thus making use of Denning's third suggested technique), and is implemented through the definition of property areThresholdsExceeded, implementing the following thresholds

$$0 \leq \text{markovChainSequenceMultiplication} \leq 10$$

$$0 \leq \text{userBehaviourAnomaly} \leq 0.9$$

The first threshold implies that the markov chain sequence multiplication must lie between 0 and 0.9, whereas the second threshold requires that the user anomaly rating using statistical moments as defined above must lie between 0 and 0.9. Failure to adhere to these thresholds results in the user connection being terminated. Note that these thresholds have been chosen as heuristics, and can be altered as required in the future. This also applies to the valuations chosen for the risk factors associated to each command.

Probabilistic Monitoring

We have currently implemented two comprehensive components, one which quantifies the system load, and another intrusion detection system based on multiple techniques. The next and final step is to use the information gathered by these components in order to develop a mechanism which probabilistically chooses users worth monitoring. Hence, this mechanism is to use the information extracted from both components in order to evaluate the final probability governing whether the user should be monitored.

Extracting information from the system profiler is trivial, since an algorithm has already been presented which evaluates the system load. However, extracting information from the intrusion detection system requires some work since statistics extracted by the intrusion detection system characterising that particular user's behaviour is not directly available at runtime upon user login. Given that both the markov chain sequence multiplication as well as the user abnormality rating quantified by the anomaly detection technique using statistical moments are both frequently sampled and dumped to database, the probabilistic monitoring algorithm loads all the dump records for a particular user, and calculates the average value of all recorded dumps for both valuations representing the degree of abnormal user behaviour. These two valuations present on average the degree of user behaviour abnormality as specified by two distinct anomaly detection techniques. Hence, both valuations are again averaged (using equal weighting for both techniques) in order to represent the final valuation representing the overall user risk factor as dictated by the intrusion detection system. this is implemented by method calculateUserRiskFactor.

Given that we have now defined all the required techniques for the quantification of both the system load and the user risk factor, all that requires is the evaluation of the final probability. This probability takes into consideration both valuations, and is expressed in the following valuation

$$(\,1\,-\,\text{systemLoad}\,)\,\times\,0.4\,+\,\text{userRiskFactor}\,\times\,0.6$$

Thus, having a low system load and/or a high user risk factor increases the probability of monitoring the user, as required. Moreover, a slight preference is given to the user risk factor, since a high risk user is to overrule the issue of system load, and should be monitored nonetheless. Given this setting, implementing the probabilistic monitoring algorithm involves the implementation of the following property

```
PROPERTY probabilisticChoice  {
  STATES  {
    BAD { }  NORMAL { }  STARTING { start }  }
  TRANSITIONS {
    start -> start[userLogin \  \ double userRiskFactor = calculateUserRiskFactor(...);
                                  double systemLoad = systemLoad();
                                  double probMonitor = (1 - systemLoad)*0.4
                                                              + userRiskFactor*0.6;
                                  Random generator = new Random();
                                  double randProb = generator.nextDouble();
```

```
                        if (randProb > probMonitor) {
                          userMonitored = false;
                          <switch off statistics>
                        }
                        <Log user login and corresponding choice
                                                              into database>
    ]
         }
}
```

Hence, upon login the user risk factor and system load are evaluated, and based on these values and the above equation the overall probability of monitoring this user is evaluated. Based on this probability the actual monitoring decision is made, whereby if the probabilistic decision is made against the monitoring of the user session, all properties and statistics are switched off. Note that user logs of each probabilistic choice upon login are also kept within the system. Also note that while we acknowledge that using the Java implementation for random number generation is only pseudo-random, we shall ignore this issue for now since this implementation is still at the proof of concept stage.

11.6 Conclusions

Intrusion detection is a field which enjoyed considerable research within the scientific community in recent years due to its vast practical applicability. This resulted in multiple techniques being developed, broadly classified under anomaly, misuse and specification based techniques, each with their own advantages. Nevertheless, as presented above the field is non-trivial and is faced with damning issues, one of which is the best rate fallacy, as well as the issue of incomplete information. It is for this reason that recent work is shifting more to the application of artificial intelligence techniques such as artificial neural networks and clustering. However, building an intrusion detection system which achieves the required effectiveness levels remains a hard task.

The application of LarvaStat to intrusion detection has potential, especially with regards to certain anomaly detection techniques since the tool allows for the expression and collection of statistics required for these techniques. We consequently presented a comprehensive case study in the form of a system capable of non-trivial tasks, including the monitoring of the system for system profiling, as well as user monitoring in the form of an intrusion detection system. Also, the implemented system acts as a probabilistic monitor, whereby the choice of monitoring the user lies with the system load upon user login, as well as the user behaviour abnormality as observed by the system during previous logins. It is indeed encouraging that this required complex functionality is written using our presented LarvaStat script without altering a single line of the underlying ftpd server code. Perhaps one criticism leveled at this case study is that certain algorithms are heuristic based, and would benefit from a formal characterisation. While indeed true, the focus of this comprehensive case study is proof of concept, symbolising the fact that complex statistical functionality required by a non-trivial field such as intrusion detection

is indeed expressible using LarvaStat. Note that the implemented intrusion detection system is not exempt from issues such as the base rate fallacy. Hence, given that the system is partly based on anomaly detection techniques, this implies a possibly non-negligible false alarm rate, which in turn drastically reduces the overall system effectiveness.

Part IV

Conclusions

12. Conclusions

The following chapter concludes the dissertation, firstly by presenting an a posteriori concise summary of the presented work. This is followed by the discussion of identified limitations, and is followed by a discussion of future work envisaged. The final section concludes the thesis by offering a few concluding thoughts regarding our work.

12.1 Summary

With the ever increasing ubiquitousness of computation, the need for reliable software grows accordingly. Reliability can be guaranteed through the certification of software correctness i.e. guarantees that the system implementation adheres to its specification. This field is commonly known as software verification, and admits multiple techniques such as testing and model checking. However, given the intractability of verifying each possible execution trace, and the lack of coverage implied by testing, we turn to an alternate approach for software verification, known as runtime verification. Runtime verification is concerned with the verification of the currently executing system trace (during the actual system execution), and can be considered as an extension to testing with augmented guarantees through the use of more powerful specification languages. However, runtime verification admits certain non-trivial issues, and include the issues of instrumentation, reparation and system overhead. The issue of system overhead is particularly crucial for practical settings, and this issue is also affects LarvaStat due to the fact that the presented statistical logics induces an overhead on the system for its computation.

We also motivated the need for the augmentation of statistics with runtime verification. It is often better to watch for indicators of failure than to wait for failure. Also, statistics allow for the straightforward specification of non-functional requirements, such as specifications related to performance and security — as evidenced by the case studies previously presented. Moreover, from a utilitarian point of view the collection of statistics over runtime executions allows for applications to fields, as evidenced by the intrusion detection case study case study implementation.

The augmentation of runtime verification with the notion of statistics implies the need for an underlying runtime verification framework. We identified this framework to be LARVA, an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs, which is sufficiently expressive for the simulation of our statistical framework, apart

from being an expressive runtime verification tool and logic worth extending.

This thesis presented an extension to LARVA with statistical capabilities, with the resulting tool and logic called LarvaStat. The development of LarvaStat took several steps, the first being the creation of the notions driving the statistical framework. This primarily resulted in the generation of the point and interval statistics, intervals, as well as the notion of the statistical event. We believe this approach to be sufficiently expressive for the specification of a wide variety of statistics. Moreover, what is also interesting about our presented approach is the fact that it effectively serves as one of the first (if not the first) approaches toward the specification of statistical properties with an additional real time component. The addition of real time within our statistical logic was primarily presented through the defined intervals, whereby both the duration and the time interval allow for the collection of statistics within time windows. In truth, any statistic can also listen to clock events defined in LARVA, which makes the approach for the integration of statistics and real time as flexible as required.

Following this, we presented the approach taken for the execution of the motivated framework through the conversion of our defined statistical constructs into DATEs, which act as the underlying mathematical framework driving LARVA. Hence, using such conversions all that is required for the execution of our framework is to convert the statistical constructs to semantically equivalent DATE constructs, whereby DATE constructs are then readily executable within LARVA.

The following step involved the design of a domain specific embedded language extending LARVA. It is through this extension that the presented statistical framework can be expressed. Note that during the design of this language care was taken so as to integrate the additional constructs seamlessly with the current LARVA script, apart from opting for simplicity and intuitive statistics specification.

Given the availability of a language capable of specifying the required statistics, the next step involved the creation of a compiler, which converts scripts written in this language into an executable artefact. However, given the existing theory for the conversion of the statistical framework into DATEs, as well as the availability of the LARVA tool, we implement this LarvaStat compiler as a converter from a LarvaStat script (hence containing LARVA constructs, as well as additional statistical constructs) into a semantically equivalent LARVA script (containing only LARVA constructs). This resulting script is then converted to an executable form by the available LARVA compiler.

The final step involved the analysis and evaluation of our framework. Analysis was executed through the definition of an operational semantics for the created statistical logic, as well as a formal definition for the logic implied by the interval definitions. Evaluation was executed through the implementation of numerous case studies, with the primary justification of LarvaStat coming from the development of a major case study implementing a probabilistic intrusion detection system and integrated system profiler. Hence, it is for this reason that Intrusion Detection was thoroughly investigated, with special emphasis placed on statistical techniques

used in the field.

12.2 Limitations

We identify a set of shortcomings with the solution presented within this dissertation, and are enlisted below.

- **Certain features are expensive** — Although we have not presented a proper characterisation of the system overhead induced by our framework, it is immediately apparent that certain features may become prohibitively expensive unless handled with care. More specifically, we recognise the two culpable features to be the specification of non-incrementally computable statistics, and the use of time intervals. This is because the specification of non-incrementally computable statistics may use an unbounded amount of memory, whereas the current execution of time intervals using polling requires very frequent updates — some (most?) of which are redundant.
- **Application to intrusion detection is insufficient** — Although the application of our work to the setting of intrusion detection was very beneficial and straightforward, we envisaged the statistical framework to be a general solution applicable to multiple fields. Hence, we require the application of our framework and tool LarvaStat to other fields requiring online statistical computations, such as user modeling, quality of service and performance profiling.
- **The current solution lacks robustness** — We feel that both the presented statistical logic as well as the developed tool LarvaStat are still at a proof of concept level. In truth this issue is related to the previous, whereby we motivate the need to apply our work to numerous industrial settings before reaching a certain level of confidence which we require of our work.

While not discounting that the presented work needs further refinement, we believe this solution to be a very reasonable step forward in the right direction with potential, especially when considering the apparent lack of alternative solutions for a general statistical framework.

12.3 Future Work

We identify three core issues which we would like to study further, these being the development of a probabilistic framework, further characterisation of intervals beyond statistical settings, and the proper analysis of the overhead incurred by our statistical framework on the underlying system.

12.3.1 A Probabilistic Framework

Analogously to the augmentation of runtime verification with statistical capabilities, we envisage the augmentation of the statistical framework with probabilistic capabilities. We cur-

rently recognise different classes of expressible probabilistic properties as discussed below.

The first class are referred to as probabilistic properties, and enable the encoding of properties such as "assertion A must be verified to be true 80% of the time". Similarly to the approach taken in [22], we could use the statistical framework to specify counts, through which probabilities could be extracted. In truth, the potential exists for the application of such properties to performance profiling or intrusion detection, stating properties such as "for the user to be deemed trustworthy, his actions should conform to a certain class of patterns 90% of the time".

Another identified class involves the probabilistic checking of properties. Such properties imply that their verification occurs on a probabilistic basis. As an example, given a system which monitors its user behaviour, each user is assigned a probability risk to the system, and based on this probability will the user be verified. Hence, a user deemed to be high risk will have a higher probability of being monitored than a low risk counterpart. In truth this approach is similar to that synthesized for the probabilistic intrusion detection case study. However, instead of manually implementing such properties, we would like a logic for their straightforward expression.

Yet another class of probabilistic properties which we identify are referred to as action oriented probabilistic properties, and are essentially an augmentation of the first class. It is envisaged that a logic for the expression of such properties is made up of two components; property specifications and guards. Obviously such property specifications can either be or not be adhered to by the system. Over time, each property builds a valuation signifying the probability with which that property is adhered to. Based on these probabilities will guards be defined, whereby guards define assertions on these probabilities and take the necessary action when broken. We may for example define a property specifying correct user behaviour, and multiple guards on the probability implied by this property, such that if the probability falls below a certain threshold the user is marked as malicious and is automatically ejected. However, if the user behaviour is exemplary (hence the associated probability is sufficiently high) we can automatically switch off the monitoring of the user due to having gained enough trust from the system.

12.3.2 Interval Characterisation

Our approach to intervals applied to the setting of runtime verification is valid and introduces interesting possibilities. We currently identify three issues which we would like to study further, as specified below.

Firstly, we recognise that although expressive and innovative, the execution of dynamic intervals moving with time can become rather computationally expensive. In fact, given the chosen execution strategy using polling, there exists a tradeoff between system overhead and simulation precision with respect to the semantics. This is because although the choice of polling duration is unbounded, and the smaller it is set the more accurate it simulates the interval, a very small duration interval also implies a considerable overhead due to constantly updating the statistic

valuation. Hence, especially given the potential applications, we would like to explore further the notion of dynamic intervals, possibly identifying alternate more efficient ways for their execution.

Related to the previous issue is the fact that the current execution of time intervals presented above is inexact with respect to the interval semantics (as previously discussed). However, whereas certain properties seem insensitive to such approximate time interval executions, other properties identify this inexactness as crucial and a deal breaker. Hence, we would also like to further characterise properties relying on time intervals, identifying the class of properties which are insensitive to this issue of approximate execution.

The final issue which we would like to study further is the application of intervals as presented within our framework beyond statistical settings. In fact, there exists practical applications for the verification of software within certain intervals. Such an example property could be "Verify system behaviour only when the system enters a critical section of its behaviour, and hence stop verification once it leaves this critical section".

12.3.3 System Overhead

Perhaps the most crucial issue omitted from this dissertation is the study of the overhead induced by the framework over the underlying system. As discussed in the runtime verification chapter, the issue of system overhead is crucial, since it can break system properties (or possibly falsely correct them). Moreover, it is crucial for the user to know the overhead induced by the features used within our framework, since the applicability of our framework given an application domain is often dependent on such information.

12.4 Concluding Thoughts

The main aim of this thesis was the development of a general framework for the online collection of statistical information over runtime executions, as well as the fruitful integration of the runtime verification and statistical frameworks. We consider this core aim to be largely satisfied, however recognising that work still needs to be done. This is especially true for the further analysis and overhead reduction of certain expensive features within our presented solution, as well as the overall characterisation of the overhead induced by our framework. In truth, although our presented solution was very adept for the application to an intrusion detection setting, the jury is still out whether it is sufficiently expressive for the application to other varied fields requiring such functionality. Practically, the solution to this unknown can only be solved by further application of our work, and results in the augmented robustness, reliability and maturity of our framework.

A. LarvaStat Compiler Usage Instructions

The following instructions entail the steps required for the execution of the LarvaStat compiler. Firstly note that the LarvaStat compiler distribution found on the cd provided contains two folders, called compiler and LarvaStat. Both folders are to be extracted to an arbitrary destination folder. The LarvaStat compiler converts an input LarvaStat script into a semantically equivalent LARVA script. The compiler itself requires a set of inputs, these being

1. The compiler verbosity. The input 'v' sets the compiler to verbose, thus outputting certain internal structures generated by the compiler, whereas the input 'n' sets the compiler to not verbose.
2. The target script location. As the name implies, the directory of the LarvaStat script. Note that any statistic definition packages required by the script must be in the same directory as the script.
3. The output directory. The destination folder, whereby the artifacts generated by the system are output in this directory.
4. The output script name. This optional parameter allows for the specification of the generated LARVA script name.

Note that the compiler generates both a LARVA script as well as additional Java class files within a folder entitled LarvaStat. These artifacts are then to be passed on to the LARVA compiler (also provided on the cd), which consequently compile the code into Java and AspectJ code. It is crucial that both the LARVA script as well as the folder are passed to the LARVA compiler. Finally, in order to execute the LarvaStat compiler

1. Enter the command prompt.
2. Navigate to the directory where the compiler was previously extracted.
3. Enter the following command "java -cp . compiler/LarvaStatCompiler <verbosity> <LarvaStat script Location> <Output directory> <LARVA script output name>"

B. Compiler Error Codes

The following is a complete list of the error codes defined for the LarvaStat compiler. Note that errors starting with 1 denote a missing construct, errors starting with 2 denote a missing reserved word, errors staring with 3 are miscellaneous errors while errors starting with 4 are identified semantic errors.

```
ERROR CODE 101 : MISSING SCRIPT CONSTRUCT: Missing { at the specified location.
ERROR CODE 102 : MISSING SCRIPT CONSTRUCT: Missing } at the specified location.
ERROR CODE 103 : MISSING SCRIPT CONSTRUCT: Missing [ at the specified location.
ERROR CODE 104 : MISSING SCRIPT CONSTRUCT: Missing ] at the specified location.
ERROR CODE 105 : MISSING SCRIPT CONSTRUCT: Missing : at the specified location.
ERROR CODE 106 : MISSING SCRIPT CONSTRUCT: Missing ( at the specified location.
ERROR CODE 107 : MISSING SCRIPT CONSTRUCT: Missing ) at the specified location.
ERROR CODE 108 : MISSING SCRIPT CONSTRUCT: Missing = at the specified location.
ERROR CODE 109 : MISSING SCRIPT CONSTRUCT: Missing ; at the specified location.
ERROR CODE 110 : MISSING SCRIPT CONSTRUCT: Missing \ at the specified location.
ERROR CODE 111 : MISSING SCRIPT CONSTRUCT: Missing -> at the specified location.

ERROR CODE 201 : MISSING RESERVED WORD: Missing 'INTERVAL' at the specified
                 location.
ERROR CODE 202 : MISSING RESERVED WORD: Missing 'GLOBAL' at the specified
                 location.
ERROR CODE 203 : MISSING RESERVED WORD: Missing 'OFF' at the specified
                 location.
ERROR CODE 204 : MISSING RESERVED WORD: Missing 'WHERE' at the specified
                 location.
ERROR CODE 205 : MISSING RESERVED WORD: Missing 'true'or 'false' at the
                 specified location.
ERROR CODE 206 : MISSING RESERVED WORD: Missing 'STATES' at the specified
                 location.
ERROR CODE 207 : MISSING RESERVED WORD: Missing 'TRANSITIONS' at the specified
                 location.
ERROR CODE 208 : MISSING RESERVED WORD: Missing 'FOREACH' at the specified
```

```
                location.
ERROR CODE 209 : MISSING RESERVED WORD: Missing 'PACKAGE' at the specified
                location.
ERROR CODE 210 : MISSING RESERVED WORD: Missing 'CONTEXT' at the specified
                location.
ERROR CODE 211 : MISSING RESERVED WORD: Missing 'OBJECTCONTEXT' at the
                specified location.

ERROR CODE 301 : MISSING IDENTIFIER: Missing identifier at the specified
                location.
ERROR CODE 302 : INCORRECT INTERVAL DEFINITION: Incorrect interval
                definition at the specified location.
ERROR CODE 303 : MISSING EVENT DEFINITION: Missing event definition at
                the specified location.
ERROR CODE 304 : MISSING CONDITION DEFINITION: Missing condition at the
                specified location.
ERROR CODE 305 : MISSING ACTION DEFINITION: Missing action definition at
                the specified location.
ERROR CODE 306 : MISSING DURATION DEFINITION: Missing time duration
                definition at the specified location.
ERROR CODE 307 : MISSING TIME DEFINITION: Missing time definition at
                the specified location.
ERROR CODE 308 : BADLY DEFINED STATISTIC: Missing : or = at the specified
                location.
ERROR CODE 309 : BADLY DEFINED STATISTIC: Missing INIT section at the
                specified location.
ERROR CODE 310 : BADLY DEFINED STATISTIC: Missing UPDATE section at
                the specified location.
ERROR CODE 311 : MISSING DECLARATION: Missing declaration at the
                specified location.
ERROR CODE 312 : MISSING LIST DEFINITION: Missing List declaration at
                the specified location.
ERROR CODE 313 : MISSING END OF FILE: Missing End Of File token at the
                specified location.
ERROR CODE 314 : DUPLICATE STATISTIC NAME: Name already used in script
                for particular  statistic.
ERROR CODE 315 : INCORRECT PARAMETER: Incorrectly defined parameter.
ERROR CODE 316 : INCORRECT NUMBER OF PARAMETERS: Incorrect amount of
                defined parameters.
ERROR CODE 317 : MISSING TEXT: Missing text at the specified location.

ERROR CODE 401 : STATISTIC DEFINITION MISMATCH: Statistic definition does
                not match with the statistic declaration.
ERROR CODE 402 : MISSING STATISTIC DEFINITION: No matching statistic
```

```
                definition found for particular statistic.
ERROR CODE 403 : MISSING INTERVAL: No matching interval definition found
                for particular statistic.
ERROR CODE 404 : WRONG STATISTIC DEFINITION ORDERING: Wrong statistic
                ordering with respect to statistic dependencies for
                particular statistic.
ERROR CODE 405 : MISSING STATISTIC DEFINITION: Cannot find
                implementation for statistic definition.
ERROR CODE 406 : CONTEXT NOT ALLOWED: Cannot define context for time
                interval statistic.
ERROR CODE 407 : BADLY DEFINED INTERVAL: Missing where clauses for
                definition of particular statistic.
ERROR CODE 408 : EXCLUSIVE UPDATE NOT PERMITTED: Exclusive update
                is only allowed for time interval statistics.
ERROR CODE 409 : MISSING OBJECT CONTEXT: Missing object context
                for particular statistic.
ERROR CODE 410 : WRONG ORDERING: Wrong construct ordering;
                Correct Ordering is
                Statistics...Properties...SubContexts.
```

C. Probabilistic Intrusion Detection Case Study Script

The following is the full script defined for the major case study presented in section 11.5, whereby this case study acts as a justification of LarvaStat through the implementation of a probabilistic intrusion detection system and integrated system profiler, while also applying statistical anomaly detection techniques. Applying this case study involves compiling this script using the LarvaStat compiler, compiling its result using the LARVA compiler, and consequently compiling the resulting Java and AspectJ code with the ftpd server implementation provided using the AspectJ compiler. The resulting Java byte code is run on a Java Virtual Machine, with the result being an ftpd server implementation whose execution is monitored by the monitor implied by the script below. Note that the script below assumes an installed MySQL database which it uses for information persistence.

```
---------------------- STATISTIC DEFINITION PACKAGE -----------------------------

PACKAGE PointStatPackage{
  POINTSTAT EventCount : Integer  {
    INIT{#EventCount#.setValue(new Integer(0));}
    UPDATE{ #EventCount#.setValue(#EventCount#.getValue() + 1); }
  }
}

--- PROBABILISTIC INTRUSION DETECTION SYSTEM WITH SYSTEM PROFILER IMPLEMENTATION ---

IMPORTS {
        import de.anomic.ftpd.*;
        import de.anomic.ftpd.serverCore.Session;
        import javax.naming.Context;
        import javax.naming.InitialContext;
        import javax.naming.NamingException;
        import java.sql.*;
        import java.io.*;
        import java.util.*;
}
STATDEFINITIONS {
```

```
        PointStatPackage.txt;
}
GLOBAL {
  VARIABLES{
    %% The master connection to the mySQL database
    Connection connection;
    Clock c = new Clock();
  }
  EVENTS {
    %% Download Events
    downloadStarting(ftpdProtocol p) = {call p.eventDownloadFilePre(arg1)}
    downloadComplete(ftpdProtocol p) = {call p.eventDownloadFilePost(arg1,arg2)}
    downloadEvent() = { downloadStarting() | downloadComplete() }

    %% Upload Events
    uploadStarting(ftpdProtocol p) = {call p.eventUploadFilePre(arg1)}
    uploadComplete(ftpdProtocol p) = {call p.eventUploadFilePost(arg1,arg2)}
    uploadEvent() = { uploadStarting() | uploadComplete() }

    %%Current Logged In Users Events
    passwordEvent(String res) = {execution ftpdProtocol p.PASS(arg1)
                                                        uponReturning (res) }
    logout() = {execution ftpdProtocol p.QUIT(arg1) }
    loginLogout() = { passwordEvent() | logout() }

    DumpToDatabase() = {c @ 5}

    serverCore_ObjectCreated() = {execution serverCore serverCore_newObj.new()}

    relevantStatisticalValuesUpdated() =
      { userDownloadsLast30Minutes_Event | userUploadsLast30Minutes_Event |
         userDownloadsLast30MinutesAverage_Event | userUploadsLast30MinutesAverage_Event }
  }

  %% --- CURRENT LOAD ------------------------------------------------------------
  %% -----------------------------------------------------------------------------

  %% Three point statistics used to calculate the total number of currently
                                                                  active downloads ---

  POINTSTAT DownloadStartingCount = PointStatPackage.EventCount[downloadStarting() \ ]

  POINTSTAT DownloadCompleteCount = PointStatPackage.EventCount[downloadComplete() \ ]

  POINTSTAT CurrentActiveDownloadCount : Integer  {
    INIT{CurrentActiveDownloadCount.setValue(new Integer(0));}
    EVENTS{downloadEvent}
    UPDATE{ CurrentActiveDownloadCount.setValue(DownloadStartingCount.getValue() -
                                                DownloadCompleteCount.getValue()); }
  }
```

```
%% Three point statistics used to calculate the total number of
                                                    currently active uploads -----

POINTSTAT UploadStartingCount = PointStatPackage.EventCount[uploadStarting() \ ]

POINTSTAT UploadCompleteCount = PointStatPackage.EventCount[uploadComplete() \ ]

POINTSTAT CurrentActiveUploadCount : Integer  {
  INIT{CurrentActiveUploadCount.setValue(new Integer(0));}
  EVENTS{uploadEvent}
  UPDATE{ CurrentActiveUploadCount.setValue(UploadStartingCount.getValue() -
                                                  UploadCompleteCount.getValue()); }
}

%% Three point statistics used to calculate the total number of users
                                                        currently logged in ----

POINTSTAT CurrentTotalUsersLoggedIn = PointStatPackage.EventCount[passwordEvent \
                                                    (res.equals("230 logged in"))]

POINTSTAT CurrentTotalUsersLoggedOut = PointStatPackage.EventCount[logout \ ]

POINTSTAT CurrentUsersLoggedIn : Integer  {
  INIT{CurrentUsersLoggedIn.setValue(new Integer(0));}
  EVENTS{loginLogout}
  UPDATE{ CurrentUsersLoggedIn.setValue(CurrentTotalUsersLoggedIn.getValue() -
                                            CurrentTotalUsersLoggedOut.getValue()); }
}

%% --- END CURRENT LOAD ------------------------------------------------------------
%% ---------------------------------------------------------------------------------

%% --- HISTORY ---------------------------------------------------------------------
%% ---------------------------------------------------------------------------------

INTERVALSTAT downloadsLast30Minutes : Integer  {
  INIT{downloadsLast30Minutes.setValue(new Integer(0));}
  EVENTS{downloadComplete()}
  LIST{true}
  INTERVAL{  TIME[1800 \  ]  }
  UPDATE[downloadsLast30Minutes.getList().add(new Integer(1));]
        { downloadsLast30Minutes.setValue(downloadsLast30Minutes.getList().size()); }
}

INTERVALSTAT uploadsLast30Minutes : Integer  {
  INIT{uploadsLast30Minutes.setValue(new Integer(0));}
  EVENTS{uploadComplete()}
  LIST{true}
  INTERVAL{  TIME[1800 \  ]  }
```

```
  UPDATE[uploadsLast30Minutes.getList().add(new Integer(1));]
        { uploadsLast30Minutes.setValue(uploadsLast30Minutes.getList().size()); }
}

INTERVALSTAT loggedInUsersLast30Minutes : Integer  {
  INIT{loggedInUsersLast30Minutes.setValue(new Integer(0));}
  EVENTS{passwordEvent}
  CONDITION{(res.equals("230 logged in"))}
  LIST{true}
  INTERVAL{  TIME[1800 \  ]  }
  UPDATE[loggedInUsersLast30Minutes.getList().add(new Integer(1));]
   { loggedInUsersLast30Minutes.setValue(loggedInUsersLast30Minutes.getList().size()); }
}

%% --- END HISTORY ------------------------------------------------------------------
%% ------------------------------------------------------------------------------------

%% Property which handles the initial database connection, as well as the dumping of
                                        current system state to the database ----------

PROPERTY handleDatabase  {
  STATES {  BAD { } NORMAL { } STARTING { start }  }
  TRANSITIONS  {
    start -> start[ serverCore_ObjectCreated()\ \
                    try{ Class.forName("com.mysql.jdbc.Driver");
                    connection = DriverManager.getConnection(
                               "jdbc:mysql://localhost:3306/ftpd-db", "root", "pass123" );
                    } catch (Exception e) { e.printStackTrace();}]
    start -> start[ DumpToDatabase()\ \
                    try {Statement stmt = connection.createStatement();
                    stmt.executeUpdate("INSERT INTO systemloadhistory
                    (DownloadsLast30Minutes, UploadsLast30Minutes,
                     UsersLoggedInLast30Minutes, TimeLogCreated,SystemLoad )   VALUES
                     ("+downloadsLast30Minutes.getValue()+","+
                        uploadsLast30Minutes.getValue()+","+
                        loggedInUsersLast30Minutes.getValue()+","+
                        System.currentTimeMillis()+","+
                        SC.calculateCurrentLoad(downloadsLast30Minutes.getValue(),
                                                uploadsLast30Minutes.getValue(),
                                             loggedInUsersLast30Minutes.getValue())+")");
                     } catch (Exception e) {e.printStackTrace();}c.reset(); ]
    }
}

FOREACH (Session s)  {
  VARIABLES{
    Clock userClock = new Clock();
    Clock userClock2 = new Clock();
    Channel userChannel = new Channel();
```

```
  %% Info loaded from database
  double uploadVarianceLast30MinMean2;
  double downloadVarianceLast30MinMean2;
  int n;

  %% boolean storing whether the user should be monitored
  boolean userMonitored = true;
}

EVENTS{
  userLogin(String result) = {execution ftpdProtocol p.PASS(arg1)
                            uponReturning (result) } where { s = p.getSession(); }
  setIdentity(String userNameParam) = {execution Session session
                              .setIdentity(userNameParam) } where { s = session; }
  userDumpToDatabase() = {userClock @ 5}
  dumpProbabilityRiskSample() = {userClock2 @ 5}

  userDownloadComplete() = {call ftpdProtocol p.eventDownloadFilePost(arg1,arg2)}
                                                    where { s = p.getSession(); }
  userUploadComplete() = {call ftpdProtocol p.eventUploadFilePost(arg1,arg2)}
                                                    where { s = p.getSession(); }
  receiveValue(ObjectPair obj) = { userChannel.receive(obj) }
                                          where {s = (Session)obj.getObject1();}

  %% --- Monitored Events ----------------------------------------------------

  passwordEventUser() = {execution ftpdProtocol p.PASS(arg1)}
                                                    where {s = p.getSession();}
  ChangeWorkingDirectoryEvent() = {execution ftpdProtocol p.CWD(arg1)}
                                                    where {s = p.getSession();}
  ChangeToParentDirectory() = {execution ftpdProtocol p.CDUP(arg1)}
                                                    where {s = p.getSession();}
  ChangePort() = {execution ftpdProtocol p.PORT(arg1)} where {s = p.getSession();
  setToPassiveEvent() = {execution ftpdProtocol p.PASV(arg1)}
                                                    where {s = p.getSession();}
  downloadFileEvent() = {execution ftpdProtocol p.RETR(arg1)}
                                                    where {s = p.getSession();}
  uploadFileEvent() = {execution ftpdProtocol p.STOR(arg1)}
                                                    where {s = p.getSession();}
  renameFromEvent() = {execution ftpdProtocol p.RNFR(arg1)}
                                                    where {s = p.getSession();}
  renameToEvent() = {execution ftpdProtocol p.RNTO(arg1)}
                                                    where {s = p.getSession();}
  removeDirectoryEvent() = {execution ftpdProtocol p.RMD(arg1)}
                                                    where {s = p.getSession();}
  makeDirectoryEvent() = {execution ftpdProtocol p.MKD(arg1)}
                                                    where {s = p.getSession();}
  PrintWorkingDirectoryEvent() = {execution ftpdProtocol p.PWD(arg1)}
                                                    where {s = p.getSession();}
  deleteFileEvent() = {execution ftpdProtocol p.DELE(arg1)}
```

```
                                                        where {s = p.getSession();}
  ListEvent() = {execution ftpdProtocol p.LIST(arg1)} where {s = p.getSession();}
  userEvent() = {execution ftpdProtocol p.USER(arg1)} where {s = p.getSession();}
  helpEvent() = {execution ftpdProtocol p.HELP(arg1)} where {s = p.getSession();}

  notRenameToEvents() = {  PrintWorkingDirectoryEvent() | ChangePort() |
        setToPassiveEvent() | helpEvent() | passwordEventUser() |
        ChangeWorkingDirectoryEvent() | ChangeToParentDirectory() | userEvent() |
        deleteFileEvent() | downloadFileEvent() | uploadFileEvent() |
        renameFromEvent()| removeDirectoryEvent() | makeDirectoryEvent() |
        ListEvent()  }
  notPasswordEvents() = {  renameToEvent() | PrintWorkingDirectoryEvent() |
       ChangePort() | setToPassiveEvent() | helpEvent() |
       ChangeWorkingDirectoryEvent() | ChangeToParentDirectory() | userEvent() |
       deleteFileEvent() | downloadFileEvent() | uploadFileEvent() |
       renameFromEvent()| removeDirectoryEvent() | makeDirectoryEvent() |
       ListEvent()  }
}

%% --- VARIANCE EVALUATION STATISTICS ------------------------------------------
%% -----------------------------------------------------------------------------

POINTSTAT totalLoginCount : Integer  {
  INIT{ totalLoginCount.setValue(new Integer(0));  }
  EVENTS{        userLogin() }
  CONDITION{ result.equals("230 logged in") }
  UPDATE{ totalLoginCount.setValue(totalLoginCount.getValue() + 1); }
}

INTERVALSTAT userDownloadsLast30Minutes : Integer  {
  INIT{userDownloadsLast30Minutes.setValue(new Integer(0));}
  EVENTS{userDownloadComplete()}
  LIST{true}
  INTERVAL{  TIME[1800 \  ]  }
  UPDATE[userDownloadsLast30Minutes.getList().add(new Integer(1));]
        { userDownloadsLast30Minutes.setValue(userDownloadsLast30Minutes
                                                    .getList().size()); }
}

POINTSTAT userDownloadsLast30MinutesVariance : Double  {
  INIT{ userDownloadsLast30MinutesVariance.setValue(new Double(0.0));  }
  EVENTS{        userDumpToDatabase }
  UPDATE{ userDownloadsLast30MinutesVariance.setValue(
            SC.calculateNextVarianceValue(downloadVarianceLast30MinMean2,
                 userDownloadsLast30MinutesAverage.getValue(), n,
                 (double)userDownloadsLast30Minutes.getValue()));
            downloadVarianceLast30MinMean2 =
                        userDownloadsLast30MinutesVariance.getValue() * (n+1);}
}
```

```
POINTSTAT userDownloadsLast30MinutesAverage : Double  {
  INIT{ userDownloadsLast30MinutesAverage.setValue(new Double(0.0)); }
  EVENTS{        userDumpToDatabase }
  UPDATE{ userDownloadsLast30MinutesAverage.setValue(
            SC.calculateNextAverageValue(userDownloadsLast30MinutesAverage
                .getValue(),n,(double)userDownloadsLast30Minutes.getValue())); }
}

INTERVALSTAT userUploadsLast30Minutes : Integer  {
  INIT{userUploadsLast30Minutes.setValue(new Integer(0));}
  EVENTS{userUploadComplete()}
  LIST{true}
  INTERVAL{  TIME[1800 \  ]         }
  UPDATE[userUploadsLast30Minutes.getList().add(new Integer(1));]
  { userUploadsLast30Minutes.setValue(userUploadsLast30Minutes.getList().size()); }
}

POINTSTAT userUploadsLast30MinutesVariance : Double  {
  INIT{ userUploadsLast30MinutesVariance.setValue(new Double(0.0)); }
  EVENTS{ userDumpToDatabase }
  UPDATE{ userUploadsLast30MinutesVariance.setValue(SC.calculateNextVarianceValue(
            uploadVarianceLast30MinMean2,userUploadsLast30MinutesAverage.getValue(),
            n,(double)userUploadsLast30Minutes.getValue()));
          uploadVarianceLast30MinMean2 = userUploadsLast30MinutesVariance.getValue()
                                                                        * (n+1);}
  }

  POINTSTAT userUploadsLast30MinutesAverage : Double  {
    INIT{ userUploadsLast30MinutesAverage.setValue(new Double(0.0)); }
    EVENTS{        userDumpToDatabase }
    UPDATE{ userUploadsLast30MinutesAverage.setValue(SC.calculateNextAverageValue(
                 userUploadsLast30MinutesAverage.getValue(),n,
                 (double)userUploadsLast30Minutes.getValue())); }
  }

  %% --- END VARIANCE EVALUATION ------------------------------------------------
  %% ----------------------------------------------------------------------------

  %% --- MARKOV CHAIN STATISTICS ------------------------------------------------
  %% ----------------------------------------------------------------------------

  POINTSTAT ProbabilityChainStat : Double  {
    INIT{ProbabilityChainStat.setValue(new Double(1));}
    EVENTS{receiveValue}
    UPDATE{ ProbabilityChainStat.setValue(ProbabilityChainStat.getValue()
                                              * ((Double)obj.getObject2())); }
  }

  %% --- END MARKOV CHAIN -------------------------------------------------------
  %% ----------------------------------------------------------------------------
```

```
PROPERTY sendRiskProbability  {
  STATES { BAD { } NORMAL { } STARTING { start } }
  TRANSITIONS  {
    start -> start[passwordEventUser \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1))); ]
    start -> start[ChangeWorkingDirectoryEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.5))); ]
    start -> start[ChangeToParentDirectory \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.5))); ]
    start -> start[ChangePort \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1.5))); ]
    start -> start[setToPassiveEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1.5))); ]
    start -> start[downloadFileEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.8))); ]
    start -> start[uploadFileEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(3))); ]
    start -> start[renameFromEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1.5))); ]
    start -> start[renameToEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1.5))); ]
    start -> start[removeDirectoryEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(4))); ]
    start -> start[makeDirectoryEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(1.5))); ]
    start -> start[deleteFileEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(4))); ]
    start -> start[PrintWorkingDirectoryEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.5))); ]
    start -> start[ListEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.5))); ]
    start -> start[helpEvent \ (userMonitored == true) \
                   userChannel.send(new ObjectPair(s,new Double(0.5))); ]
  }
}

PROPERTY areThresholdsExceeded  {
  STATES  {  BAD { }  NORMAL { }  STARTING { start }  }
  TRANSITIONS  {
    start -> start[ProbabilityChainStat_Event\(userMonitored == true) \
                   if (ProbabilityChainStat.getValue().doubleValue() > 10){
                    try{ DataOutputStream dos = new DataOutputStream(
                                        s.controlSocket.getOutputStream());
                    dos.writeBytes("SUSPICION LEVEL EXCEEDED:
                                             Terminating Connection...");
                    s.controlSocket.close(); } catch (Exception e){
                    e.printStackTrace(); } } ]
    start -> start[ relevantStatisticalValuesUpdated \ (userMonitored == true) \
                   if ((SC.calculateIntrusionProbabilityVariance(
```

```
                        totalLoginCount.getValue(),
                        userDownloadsLast30MinutesVariance.getValue(),
                        userUploadsLast30MinutesVariance.getValue(),
                        userDownloadsLast30Minutes.getValue(),
                        userUploadsLast30Minutes.getValue(),
                        userDownloadsLast30MinutesAverage.getValue(),
                        userDownloadsLast30MinutesAverage.getValue()) >= 0.9)
                        && (totalLoginCount.getValue() >= 10)  ) {
                    try{ DataOutputStream dos = new DataOutputStream(
                                    s.controlSocket.getOutputStream());
                    dos.writeBytes("SUSPICION LEVEL EXCEEDED:
                                                Terminating Connection...");
                    s.controlSocket.close(); }
                    catch (Exception e){ e.printStackTrace(); } }   ]
  }
}

PROPERTY sampleProbabilityRiskBehaviour  {
  STATES {  BAD { } NORMAL { } STARTING { start }  }
  TRANSITIONS {
    start -> start[ dumpProbabilityRiskSample \ (userMonitored == true) \
                  try {Statement stmt = connection.createStatement();
                  stmt.executeUpdate("INSERT INTO mcprobabilityrisk (
                            userName,probabilityRisk,TimeCreated ) VALUES
                            ('"+s.identity+"',"+SC.calculateProbabilityRisk(
                            ProbabilityChainStat.getValue())+","+
                            System.currentTimeMillis()+")");
                  } catch (Exception e) {e.printStackTrace();}
                  userClock2.reset(); ]
  }
}

%% --- Property which probabilistically decides upon user login whether to
   monitor this login session or not, and logs this choice ---------------
PROPERTY probabilisticChoice  {
  STATES {  BAD { } NORMAL { } STARTING { start }  }
  TRANSITIONS {
    start -> start[ userLogin \  \
                    double userRiskFactor = SC.calculateUserRiskFactor(
                                                connection,s.identity);
                    double systemLoad = SC.systemLoad(connection,
                    CurrentActiveDownloadCount.getValue(),
                    CurrentActiveUploadCount.getValue(),
                    CurrentUsersLoggedIn.getValue());
                    double probMonitor = (1 - systemLoad)*0.4 +
                                                  userRiskFactor*0.6;
                    Random generator = new Random();
                    double randProb = generator.nextDouble();
                    if (randProb > probMonitor) { userMonitored = false;
                          totalLoginCount.switchOff();
```

```
                            userDownloadsLast30Minutes.switchOff();
                            userDownloadsLast30MinutesVariance.switchOff();
                            userDownloadsLast30MinutesAverage.switchOff();
                            userUploadsLast30Minutes.switchOff();
                            userUploadsLast30MinutesVariance.switchOff();
                            userUploadsLast30MinutesAverage.switchOff();
                            ProbabilityChainStat.switchOff(); }
                            try {Statement stmt
                                          = connection.createStatement();
                              stmt.executeUpdate("INSERT INTO probabilistiicchecklog
                              (TimeLoginAttempted,userName,clientIp,
                               propertiesChecked,SystemLoad,RiskFactor,
                               ProbabilityCheck) VALUES
                               ("+System.currentTimeMillis()+",'"+s.identity+
                               "','"+s.userAddress.getHostAddress()+"',"+
                               userMonitored+","+systemLoad+","+
                               userRiskFactor+","+probMonitor+")"); }
                            catch (Exception e) {e.printStackTrace();} ]
  }
}

PROPERTY userHandleDatabase {
  STATES {  BAD { } NORMAL { } STARTING { start }  }
  TRANSITIONS {
    start -> start[setIdentity \ (userMonitored == true) \
                   totalLoginCount.setValue(
                   SC.returnUserLoginCount(connection,userNameParam));
                   userUploadsLast30MinutesAverage.setValue(
                         SC.returnUserUploadsLast30MinutesAverage(
                         connection,userNameParam));
                   userUploadsLast30MinutesVariance.setValue(
                         SC.returnUserUploadsLast30MinutesVariance(
                         connection,userNameParam));
                   userDownloadsLast30MinutesAverage.setValue(
                         SC.returnUserDownloadsLast30MinutesAverage(
                         connection,userNameParam));
                   userDownloadsLast30MinutesVariance.setValue(
                         SC.returnUserDownloadsLast30MinutesVariance(
                         connection,userNameParam));
                   uploadVarianceLast30MinMean2 =
                         SC.returnUserUploadsLast30MinutesMean2(
                         connection,userNameParam);
                   downloadVarianceLast30MinMean2 =
                         SC.returnUserDownloadsLast30MinutesMean2(
                         connection,userNameParam);
                   n = SC.returnCount(connection,userNameParam); ]
    start -> start[userDumpToDatabase \ (userMonitored == true) \
                   n++; try {Statement stmt = connection.createStatement();
                   stmt.executeUpdate("INSERT INTO userinformation
                      (UserName,NoOfLogins,UploadsLast30MinutesVariance,
```

```
                        UploadsLast30MinutesMean2,UploadsLast30MinutesAverage,
                        DownloadsLast30MinutesVariance,DownloadsLast30MinutesMean2,
                        DownloadsLast30MinutesAverage,LastIPUsed,TimeUpdated,
                        varianceRiskProbability ) VALUES
                        ('"+s.identity+"',"+totalLoginCount.getValue()+","+
                        userUploadsLast30MinutesVariance.getValue()+","+
                        uploadVarianceLast30MinMean2+","+
                        userUploadsLast30MinutesAverage.getValue()+","+
                        userDownloadsLast30MinutesVariance.getValue()+","+
                        downloadVarianceLast30MinMean2+","+
                        userDownloadsLast30MinutesAverage.getValue()+",'"+
                        s.userAddress.getHostAddress()+"',"+
                        System.currentTimeMillis()+","+
                        SC.calculateIntrusionProbabilityVariance(
                          totalLoginCount.getValue(),
                          userDownloadsLast30MinutesVariance.getValue(),
                          userUploadsLast30MinutesVariance.getValue(),
                          userDownloadsLast30Minutes.getValue(),
                          userUploadsLast30Minutes.getValue(),
                          userDownloadsLast30MinutesAverage.getValue(),
                          userDownloadsLast30MinutesAverage.getValue())+")");
                    } catch (Exception e) {e.printStackTrace();} userClock.reset(); ]
  }
}

%% --- MONITORING PROPERTIES ------------------------------------------------------
%% --------------------------------------------------------------------------------

%% Property as defined in RFC959 Protocol
PROPERTY renameSequence{
  STATES {  BAD {sequenceNotAdheredTo } NORMAL { sequenceMiddle }
            STARTING { sequenceStart }  }
  TRANSITIONS {
    sequenceStart -> sequenceMiddle[renameFromEvent()\ (userMonitored == true) \]
    sequenceMiddle -> sequenceStart[renameToEvent()\ \]
    sequenceMiddle -> sequenceNotAdheredTo [notRenameToEvents \ \
                            try{ DataOutputStream dos = new DataOutputStream(
                                          s.controlSocket.getOutputStream());
                            dos.writeBytes("RENAMESEQUENCE PROPERTY VIOLATED:
                                                  Terminating Connection...");
                            s.controlSocket.close();
                            } catch (Exception e){ e.printStackTrace(); } ]
  }
}

PROPERTY userPass{
  STATES {  BAD {sequenceNotAdheredTo } NORMAL { sequenceMiddle }
            STARTING { sequenceStart }  }
  TRANSITIONS  {
    sequenceStart -> sequenceMiddle[userEvent()\ (userMonitored == true) \]
```

```
          sequenceMiddle -> sequenceStart[passwordEventUser()\ \]
          sequenceMiddle -> sequenceNotAdheredTo [notPasswordEvents\ \
                                try{ DataOutputStream dos = new DataOutputStream(
                                                s.controlSocket.getOutputStream());
                                dos.writeBytes("USERPASS PROPERTY VIOLATED:
                                                Terminating Connection...");
                                s.controlSocket.close();
                                } catch (Exception e){ e.printStackTrace(); } ]
        }
      }

      PROPERTY deleteFilePermission{
        STATES {  BAD { noPermission } NORMAL {} STARTING { legalStatus } }
        TRANSITIONS  {
          legalStatus -> legalStatus [deleteFileEvent\ ((userMonitored == true) &&
                  (ftpdPermissions.permissionWrite(s.getIdentity()) == true )) \  ]
          legalStatus -> legalStatus [deleteFileEvent\ ((userMonitored == true) &&
                  (ftpdPermissions.permissionWrite(s.getIdentity()) == false )) \
                  try{ DataOutputStream dos = new DataOutputStream(
                                          s.controlSocket.getOutputStream());
                  dos.writeBytes("DELETEFILEPERMISSION PROPERTY VIOLATED:
                                                  Terminating Connection...");
                 s.controlSocket.close(); } catch (Exception e){
                                                    e.printStackTrace(); }  ]
       }
      }

      PROPERTY listDirectoryPermission{
        STATES {  BAD { noPermission } NORMAL {} STARTING { legalStatus }  }
        TRANSITIONS {
          legalStatus -> legalStatus [ListEvent\ ((userMonitored == true) &&
                    (ftpdPermissions.permissionRead(s.getIdentity()) == true )) \  ]
          legalStatus -> legalStatus [ListEvent\ ((userMonitored == true) &&
                    (ftpdPermissions.permissionRead(s.getIdentity()) == false )) \
                    try{ DataOutputStream dos = new DataOutputStream(
                      s.controlSocket.getOutputStream());
                      dos.writeBytes("LISTDIRECTORYPERMISSION PROPERTY VIOLATED:
                      Terminating Connection..."); s.controlSocket.close(); }
                    catch (Exception e){ e.printStackTrace(); }  ]
        }
     }

    %% --- END MONITORING PROPERTIES ------------------------------------------
    %% ------------------------------------------------------------------------
  }
}
METHODS {
  public class SC {

  public static double calculateProbabilityRisk(Double input) {
```

```
  if (input >= 10)
    return 1;
  else
    return input / 10;
}

public static double calculateAverageProbabilityRiskBehaviour(
                                                Connection c,String userName) {
  double res = 0;
  try {
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                              mcprobabilityrisk WHERE userName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    double sum = 0;

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                             mcprobabilityrisk WHERE userName = '"+userName+"'");
      while(results2.next())  {
        sum += results2.getDouble("probabilityRisk");
      }
      results2.close();
      res = sum / rowCount;
    }
    else
      res = 1;
  }
  catch (Exception e)  { e.printStackTrace(); }

  return res;
}

public static double calculateAverageProbabilityRiskVariance(Connection c,
                                                              String userName) {
  double res = 0;
  try {
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                             userinformation WHERE userName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    double sum = 0;

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                            userinformation WHERE userName = '"+userName+"'");
```

```
      while(results2.next()) {
        sum += results2.getDouble("varianceRiskProbability");
      }
      results2.close();
      res = sum / rowCount;
    }
    else
      res = 1;
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

public static double calculateUserRiskFactor(Connection c,String userName) {
  double probRiskBehaviour =
                        calculateAverageProbabilityRiskBehaviour(c,userName);
  double intrusionDetectionProb =
                        calculateAverageProbabilityRiskVariance(c,userName);

  return 0.5*probRiskBehaviour + 0.5*intrusionDetectionProb;
}

public static double calculateIntrusionProbabilityVariance(int noOfLogins,
  double downloadVariance, double uploadVariance, double dlLast30Min,
  double upLast30Min,double overallDownload30MinAverage,
  double overallUpload30MinAverage)  {
  // 10 logins are taken in order to build a sufficient idea of the
                                        user behaviour's characteristics.
  if (noOfLogins >= 10) {
    double downloadRiskFactor = 0;
    double uploadRiskFactor = 0;

    if (downloadVariance <= 20)  {
      if (overallDownload30MinAverage < dlLast30Min) {
        downloadRiskFactor = 1 - Math.exp(-((dlLast30Min
                                      - overallDownload30MinAverage)/5));
      }
    }

    if (uploadVariance <= 20) {
      if (overallUpload30MinAverage < upLast30Min) {
        uploadRiskFactor = 1 - Math.exp(-((upLast30Min
                                       - overallUpload30MinAverage)/5));
      }
    }

    if (downloadRiskFactor > uploadRiskFactor)
      return downloadRiskFactor;
```

```
    else
      return uploadRiskFactor;
  }
  else
    return 1;
}

static Integer returnCount(Connection c,String userName) {
  Integer res = 0;

  try  {
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                        userinformation WHERE UserName = '"+userName+"'");
    results.next();
    res = results.getInt(1);
  }
  catch (Exception e) {
    e.printStackTrace();
  }

  return res;
}

static Double returnUserDownloadsLast30MinutesAverage(Connection c,
                                                      String userName) {
  Double res = 0.0;

  try{
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                      userinformation WHERE UserName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                     userinformation WHERE UserName = '"+userName+"'");
      long timeCreated = 0;

      while (results2.next()) {
        long time = results2.getLong("TimeUpdated");

        if (timeCreated < time) {
          timeCreated = time;
          res = results2.getDouble("DownloadsLast30MinutesAverage");
        }
      }
```

```
      results2.close();
    }
  }
      catch (Exception e) { e.printStackTrace(); }

  return res;
}

static Double returnUserUploadsLast30MinutesAverage(Connection c,
                                                    String userName) {
  Double res = 0.0;

  try{
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                    userinformation WHERE UserName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    if (rowCount != 0)  {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
        userinformation WHERE UserName = '"+userName+"'");
      long timeCreated = 0;

      while (results2.next()) {
        long time = results2.getLong("TimeUpdated");

        if (timeCreated < time)  {
          timeCreated = time;
          res = results2.getDouble("UploadsLast30MinutesAverage");
        }
      }
      results2.close();
    }
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static Double returnUserUploadsLast30MinutesVariance(Connection c,
                                                     String userName) {
  Double res = 0.0;

  try{
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
      userinformation WHERE UserName = '"+userName+"'");
```

```
      results.next();
      int rowCount = results.getInt(1);
      results.close();

      if (rowCount != 0) {
        Statement stmt2 = c.createStatement();
        ResultSet results2 = stmt2.executeQuery("SELECT * FROM
          userinformation WHERE UserName = '"+userName+"'");
        long timeCreated = 0;

        while (results2.next()) {
          long time = results2.getLong("TimeUpdated");

          if (timeCreated < time) {
            timeCreated = time;
            res = results2.getDouble("UploadsLast30MinutesVariance");
          }
        }
        results2.close();
      }
    }
    catch (Exception e) { e.printStackTrace(); }

    return res;
  }

  static Double returnUserDownloadsLast30MinutesVariance(Connection c,
                                                         String userName) {
    Double res = 0.0;

    try{
      Statement stmt = c.createStatement();
      ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                           userinformation WHERE UserName = '"+userName+"'");
      results.next();
      int rowCount = results.getInt(1);
      results.close();

      if (rowCount != 0) {
        Statement stmt2 = c.createStatement();
        ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                          userinformation WHERE UserName = '"+userName+"'");
        long timeCreated = 0;

        while (results2.next()) {
          long time = results2.getLong("TimeUpdated");

          if (timeCreated < time) {
            timeCreated = time;
            res = results2.getDouble("DownloadsLast30MinutesVariance");
```

```
        }
      }
      results2.close();
    }
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static Double returnUserDownloadsLast30MinutesMean2(Connection c,
                                                    String userName) {
  Double res = 0.0;

  try{
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                        userinformation WHERE UserName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                             userinformation WHERE UserName = '"+userName+"'");
      long timeCreated = 0;

      while (results2.next()) {
        long time = results2.getLong("TimeUpdated");

        if (timeCreated < time)  {
          timeCreated = time;
          res = results2.getDouble("DownloadsLast30MinutesMean2");
        }
              }
      results2.close();
    }
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static Double returnUserUploadsLast30MinutesMean2(Connection c,
                                                  String userName) {
  Double res = 0.0;

  try{
    Statement stmt = c.createStatement();
```

```
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                            userinformation WHERE UserName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                              userinformation WHERE UserName = '"+userName+"'");
      long timeCreated = 0;

      while (results2.next()) {
        long time = results2.getLong("TimeUpdated");

        if (timeCreated < time) {
          timeCreated = time;
          res = results2.getDouble("UploadsLast30MinutesMean2");
        }
      }
      results2.close();
    }
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static Integer returnUserLoginCount(Connection c,String userName) {
  Integer res = 0;

  try{
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                          userinformation WHERE UserName = '"+userName+"'");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    if (rowCount != 0) {
      Statement stmt2 = c.createStatement();
      ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                         userinformation WHERE UserName = '"+userName+"'");
      long timeCreated = 0;

      while (results2.next()) {
        long time = results2.getLong("TimeUpdated");

        if (timeCreated < time)  {
          timeCreated = time;
```

```
          res = results2.getInt("NoOfLogins");
        }
      }
      results2.close();
    }
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static double calculateCurrentLoad(Integer currentDownloads,
                  Integer currentUploads,Integer currentLoggedInUsers) {
  int currentTransfers = currentDownloads + currentUploads;

  if (currentTransfers > 20) {
    double transfersPerUser = ((double)currentTransfers /
                                                    currentLoggedInUsers);
    if (transfersPerUser > 5)
      return 1;
    else
      return (transfersPerUser / 5);
  }
  else
    return 0;
}

static double calculateHistorySystemLoad(Connection c) {
  double res = 0;

  try {
    Statement stmt = c.createStatement();
    ResultSet results = stmt.executeQuery("SELECT count(*) FROM
                                                     systemloadhistory");
    results.next();
    int rowCount = results.getInt(1);
    results.close();

    int counter;
    if (rowCount < 24)
      counter = 0;
    else
      counter = rowCount - 24;

    int tempNum = 0;

    double lastHourCount = 0;
    double lastSixHoursCount = 0;
    double lastTwelveHoursCount = 0;
    int lastHourNum = 0;
```

```
    int lastSixHoursNum = 0;
    int lastTwelveHoursNum = 0;

    Statement stmt2 = c.createStatement();
    ResultSet results2 = stmt2.executeQuery("SELECT * FROM
                                              systemloadhistory");

    while (counter > tempNum) {
      results2.next();
      tempNum++;
    }
    while (results2.next()) {
      lastTwelveHoursCount += results2.getDouble(6);
      lastTwelveHoursNum++;

      if (tempNum >= (rowCount - 12)) {
        lastSixHoursCount += results2.getDouble(6);
        lastSixHoursNum++;
      }

      if (tempNum >= (rowCount - 2)) {
        lastHourCount += results2.getDouble(6);
        lastHourNum++;
      }

      tempNum++;
    }

    res = 0.75*(lastHourCount / lastHourNum) + 0.15*(
                lastSixHoursCount / lastSixHoursNum) +
                0.1*(lastTwelveHoursCount / lastTwelveHoursNum);
    results.close();
  }
  catch (Exception e) { e.printStackTrace(); }

  return res;
}

static double systemLoad(Connection c,Integer currentDownloads,
           Integer currentUploads,Integer currentLoggedInUsers) {
  return calculateHistorySystemLoad(c)*0.4 +
     calculateCurrentLoad(currentDownloads,currentUploads,
     currentLoggedInUsers)*0.6;
}

public static double calculateNextAverageValue(
                     double currentAverage,int n,double newValue) {
  return currentAverage + (newValue - currentAverage) / (n + 1);
}
```

```
  public static double calculateNextVarianceValue(double mean2,
                       double currentAverage,int n,double newValue) {
    int newN = n + 1;
    double delta = newValue - currentAverage;
    double newAverage = currentAverage + delta / newN;
    double mean2New = mean2 + delta*(newValue - newAverage);

    return mean2New / newN;
  }
 }
}
```

Bibliography

[1] Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Electron. Notes Theor. Comput. Sci.*, 197(1):45–58, 2008.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205, August 2000.

[4] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Department Of Computer Engineering, Chalmers University of Technology, March 2000.

[5] Robert F. Barnes. Interval temporal logic: A note. Springer, November 1981.

[6] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. pages 44–57. Springer, 2004.

[7] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, November 2002.

[8] Zhou Chaochen, C.A.R Hoare, and Anders P.Ravn. A calculus of durations. *Oxford University Computing Laboratory, Programming Research Group*, 1991.

[9] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[10] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, The MIT Press Massachusetts Insititute Of Technology Cambridge, Massachusetts 02142, 1999.

[11] Séverine Colin and Leonardo Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, pages 525–555. Springer, 2004.

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

[13] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[14] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE Computer Society Press, June 2005.

[15] Pallab Dasgupta, P. P. Chakrabarti, Jatindra Kumar Deka, and Sriram Sankaranarayanan. Min-max computation tree logic. *Artif. Intell.*, 127(1):137–162, 2001.

[16] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13:222–232, 1987.

[17] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting statistics over runtime executions. In *In Proceedings of Runtime Verification (RV02) [1]*, pages 36–55. Elsevier, 2002.

[18] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. In *In Proceedings of Runtime Verification (RV01) [1*, pages 44–60, 2001.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[20] Klaus Havelund, John Penix, and Willem Visser, editors. *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*. Springer, 2000.

[21] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Trans. Softw. Eng.*, 19(9):886–901, 1993.

[22] Jane Jayaputera, Iman Poernomo, and Heinz Schmidt. Runtime verification of timing and probabilistic properties using wmi and .net. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 100–106, Washington, DC, USA, 2004. IEEE Computer Society.

[23] J.Postel and J.Reynolds. *FILE TRANSFER PROTOCOL (FTP)*. Network Working Group, rfc 959 edition, October 1985.

[24] Wolfram Kahl, Christopher K. Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In *8th International Conference on Relational Methods in Computer Science, RelMiCS 8*, volume 3929, pages 147–160, 2005.

[25] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[27] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. In *In Runtime Verification 2001, volume 55 of ENTCS*. Elsevier Science Publishers.

[28] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, third edition, November 1997.

[29] N. Markey, Ph. Schnoebelen, and Lab Spcification. Model checking a path (preliminary report. In *In 14th Int. Conf. Concurrency Theory, Lecture Notes in Computer Science 2761*, pages 251–265. Springer, 2003.

[30] Robert Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., 1989.

[31] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A temporal logic based framework for intrusion detection. In *In Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE*, 2004.

[32] Gordon Pace, Christian Colombo, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, LNCS 4916. Springer-Verlag, 2008.

[33] We Pose, Nicholas A. Merriam, Andrew M. Dearden, and Michael D. Harrison. Theorem proving. In *University of Glasgow, UK*, 1995.

[34] Murray R.Spiegel and Larry J.Stephens. *Statistics*. McGraw Hill, 4 edition, 2007.

[35] R. Sekar. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *In USENIX Security Symposium*, pages 63–78, 1999.

[36] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Course Technology, February 2005.