Practical Runtime Monitoring with Impact Guarantees of Java Programs with Real-Time Constraints

Christian Colombo

Faculty of ICT

University of Malta

September 2008

Submitted for the degree of Masters of Science

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

> Practical Runtime Monitoring with Impact Guarantees of Java Programs with Real-time Constraints

submitted is my work, except where acknowledged and referenced.

Christian Colombo

September 2008

Acknowledgements

I would like to sincerely thank Gordon J. Pace for his continuous support and enthusiasm which kept me going for the whole year. He also introduced me to the subject and through frequent meetings he inspired me and helped me in difficulties. Here I would like to acknowledge that parts of the work presented in this thesis are Gordon's work. These include parts of the mathematical framework of DATEs, and parts of the proofs regarding truth preservation of real-time properties.

A considerable part of this work has been done at the University of Oslo and hence I would like to thank Gerardo Schneider for his support and availability during the three months I spent in Norway.

This work would not have been possible without the availability of other researchers who were available to answer questions about their work in the field. Thus, I would like to thank Jochen Hoenicke for his help in the translation of duration calculus into automata, Joseph Cordina for his help about networks, and Anders Ravn for directing me to the right source of information. Other researchers who offered their support were Irem Aktug, Oleg Sokolsky, Johan Linde, Grigore Roşu, Feng Chen, and César Sanchez.

I would like to thank my work mates Stephen Fenech, Christian Tabone, and Claudia Borg for their presence during the research. The work which regards the translation of the contract language into Larva is mostly the work of Stephen Fenech.

I would also like to thank my family, and friends, especially Stephanie Mamo Portelli, for their continued support.

The research work disclosed in this publication is partially funded by Malta Government Scholarship Scheme grant number ME 367/07/29.

Abstract

With the ever growing need of robust and reliable software, formal methods are increasingly being employed. A recurrent problem is the intractability of exhaustively verifying software. Due to its scalability to real-life systems, testing has been used extensively to verify software. However, testing usually lacks coverage. Runtime verification is a compromise whereby the current execution trace is verified during runtime. Thus, it scales well without loss of coverage.

In this thesis we examine closely the work in runtime verification and identify potential improvements with regards to the specification of properties and guarantees which are given. For the first issue of specification we make use of a real-life case study to better understand the day-to-day properties which system architects would need to formalise. DATE is the logic which results from these experiments and experiences. It is shown to be highly expressive and versatile for real-life scenarios. This is further confirmed by the second case study carried out and presented in this work. Our logic requires a complete architecture to be usable with Java programs. This architecture, called Larva, is developed using Java and AspectJ with the aim of being able to automatically create and instrument monitoring code from a Larva script.

A central aspect of this thesis is the work on real-time properties. Such properties are very sensitive to overheads, because overheads slow down the system and use up resources. For this purpose, we present a theory based on the prominent real-time logic, duration calculus, to be able to give guarantees on the effects of slowing down/speeding up the target system due to the monitoring overhead. Another form of guarantee which we can give on real-time properties is the upperbound of memory overhead used during runtime verification. This is achieved by starting from the subset of duration calculus, QDDC, and translating it into Lustre.

To relate Larva to other tools in the field we use a benchmark to compare expressivity and resource consumption of Larva to other prominent tools. Furthermore, a subset of duration calculus (counterexample traces), QDDC, and Lustre have been shown to be translatable into Larva. This has two main purposes: first that this allows users familiar with other logics to utilise Larva, and second that the guarantees enjoyed by these logics, will also benefit Larva.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Aim and Motivation

1.1.1 Background

As computer systems become increasingly present in all the aspects of our lives — be it in transport systems, medical equipment, or online billing systems — it becomes increasingly important to provide reliable and robust software. Faults in security-critical systems can at best cost a lot of money, at worst the loss of human lives.

Thus far, the most common approach to address this problem was testing. Despite the effectiveness of proper testing, it is extremely difficult to test huge software products in a sufficiently thorough manner so as to ensure their correctness. The reason is that testing usually lacks coverage. A system does not work in a vacuum, but rather in an environment. Usually, we cannot predict (and much less simulate) all the environment behaviours to test our system in every possible situation. Another approach to provide secure software is model checking. In this case, we try to verify all execution paths which the system can possibly run into. However, this is usually impractical on a system which is large enough to be of any practical use.

Therefore, it seems that we need to find another way of ensuring that computer software is reliable and robust. This is a sort of trade-off between testing and model-checking. Taking the scalability of testing and the coverage of model-checking one can verify an execution trace during the actual runtime of a system. The idea is that we have a clear description of the system's specification representing all the acceptable behaviour and while the system is executing, one is continually ensuring that the behaviour is adhering to the specification. In this way we would be guaranteeing that whatever the environment or the input, the behaviour is still correct. In case a property violation is encountered, the verifying system can either raise an alarm or else, even more appropriately, make some action which corrects the state of the monitored system.

1.1.2 Broad Aims and Achievements

In order to provide a specification, we need to have succinct and clear notation which is not error prone itself. A number of formal notations have been proposed, each of which were designed with a particular domain in mind. Such a notation can still be difficult for a developer without a background in formal notation, so we need to find easier and clearer ways of specifying the security properties. In this work we propose the use of an automata-based logic called DATE. This logic was designed with the aim of providing all the necessary expressivity which a user may need. For example, a user should be able to capture system events and keep track of the security state of the system using states and variables. The variables and the objects associated with the events can be manipulated using specific actions on the transitions of a DATE automaton. The automaton may also trigger events based on timers. This is particularly useful for monitoring real-time properties. There are various other features which make this language particularly useful for monitoring Java programs.

However, there still remains the problem of inserting the monitoring code which takes note of the events occurring in the security-critical system and then verifies them against the specification. We need to keep the process of injecting this code also error-free and hence we need to make it automatic. To this end, we propose the use of aspect-oriented programming which injects Java monitoring code automatically in identifiable points (join points) in the system code. The most common join points are method calls; but there are other useful ones, such as, exception throws and exception handlers. Together with the DATE logic we have implemented a system Larva, to create and integrate the monitoring code with the target system.

Finally, we need to consider the problem that upon injecting monitoring code, we are also changing the system itself (slowing it down), because we have introduced an overhead to the actual system. This issue requires careful consideration because we do not want to introduce errors while trying to eliminate them. This applies more specifically to real-time systems because these are more error prone due to their strict timings. The overheads are two-fold: memory overheads and processing overheads (which slow down the system). To tackle these issues, we take two approaches: one which guarantees an upperbound of memory resources and another which ensures that slowing down (or speeding up) a real-time system will not cause the violation of the system properties which would hold on the non-monitored version.

To guarantee memory upperbounds, we decided to use the synchronous language Lustre [46], with the advantage of being able to calculate the memory required at compile time. If the security properties can be represented in Lustre, then the problem of guaranteeing memory upperbound is resolved. Lustre has already been used to monitor QDDC [45], which is a subset of duration calculus — a highly expressive dense-time logic. The advantage of duration calculus is that it is based on intervals and not on points in time. For runtime verification this has a significant advantage that there is no distinction between past and future. For this reason and its sound mathematical background, duration calculus is ideal to represent real-time software properties. Lustre is known to be easily translatable into symbolic automata and hence it was relatively easy to convert it into DATE automata. Using the conversion from QDDC to Lustre and from Lustre to LARVA, we thus provide the framework to guarantee an upperbound for the memory overhead required to monitor Java programs.

Duration calculus is too expressive to be fully implemented, but a useful subset called *implementables* [78, 84] provides enough expressive power for commonly known practical applications. This subset has been converted into phase event automata [58] and the latter did not prove to be very difficult to convert into LARVA. Apart from QDDC and the implementables, duration calculus has other interesting subsets which we have studied. In fact, there are two subsets which we will refer to as *slowdown truth preserving properties* and *speedup truth preserving properties*. For the slowdown truth preserving properties we guarantee that if such a property holds on a particular execution of a program, it will also hold on a slowed-down version of the same execution. Similarly, for the speedup truth preserving properties we guarantee that if such a property holds on a particular execution of a program, it will also hold on a speeded-up version of the same execution. These two subsets intersect for a number of useful properties which are concerned with the number of events. The number of events is not affected by slowing down or speeding up of an execution. Although parts of the slowdown/speedup subsets are not in the implementables subset, it is still useful to identify the subset of implementables which intersects with the slowdown and speedup subsets.

1.2 Practical Implications

We envision two main practical applications for which LARVA can be used: (i) during software development — connecting the implementation with requirements, and (ii) when deploying fully developed software — being either in-house or third party software. In a software development environment, the requirements of the software can be expressed in a formal notation very early in the development life cycle. If these are written using the LARVA language (or any other supported notation), eventually, during the actual implementation, the requirements can be directly related to concrete system events such as method calls or exception throws. With this little extra overhead, the monitoring of system requirements comes with no additional human intervention — just a simple compilation which includes the monitoring code generated by the LARVA system. The extra advantage of using LARVA is the proved guarantees that can be given when running the system with the monitoring code. When deploying a fully developed system, the user can decide either to keep the monitoring system (if it was already introduced), or create the monitoring system from scratch (even if the software is third party software). This freedom of removing/introducing monitoring code in real-time systems can only be guaranteed to have no impact on the target system if the monitored properties happen to lie within the subsets of slowdown/speedup properties. For example, removing the monitors from a system will usually cause the system to run faster. If the properties being verified are speedup truth preserving, we can guarantee that any property which was satisfied by the system, will still hold in the version without the monitors. Similarly, introducing monitors for slowdown properties to software will not affect the satisfaction of such properties.

1.3 Other Accomplishments

To assess whether our work is useful in practice, we have applied it to two real-life case studies: a transaction system and an intrusion detection system. The former is a system taken from a local industrial company which processes a large volume of transactions per hour. The emphasis of this case study is the integration of the Larva runtime verification architecture to the target system. On the other hand, the intrusion detection system was purposely created for the second case-study. The emphasis in this case was on the encoding and monitoring of real-time properties.

The related work in the area of runtime verification is very important in this thesis because it is useless to create new logics and architectures without showing the advantages and improvements on the existing work in the field. This was mainly achieved through a benchmark which has the aim of comparing the expressivity and resource usage of various state-of-the-art logics and tools in the field of runtime verification. To compare expressivity we selected a number of important features and for each tool we checklisted the list of features which it offers. Comparing the resource usage of the various tools was more intriguing. We created a fictitious scenario to be used as a benchmark and tried different tools to solve the same problem.

As a by-product of using other logics with our architecture to gain more guarantees, we have ultimately created a complete runtime verification infrastructure in which the users have a choice of five different notations: Larva, counterexample traces, phase event automata, QDDC, and Lustre. This makes our tool more easily usable by users who are used to the currently used notations.

1.4 Document Outline

The document is organised in five main parts as follows:

- The first one gives the necessary background for the work. This has two main components: Chapter 2 which gives a complete overview about dynamic analysis with special emphasis on runtime verification followed by

Chapter 3 which introduces the real-time logics which will feature later on in the work.

- The second part features the DATE logic and LARVA architecture. The first chapter explains the design choices and mathematics behind the DATE logic. The following two chapters explain the reasons behind the choice of syntax for the logic and the implementation details of the LARVA infrastructure.

- The third part gives the theory and proofs which relate DATE to other logics and derive guarantees on runtime verification properties. Chapter 7 gives the relationship of DATE to two other logics — counterexample traces and QDDC — proving the translations involved. The following chapter gives the theory of truth preservation of real-time properties written in duration calculus.

- The fourth part gives an account of the experiments carried using LARVA: Chapter 9 gives an account of two real-life scenarios while Chapter 10 compares LARVA to related work in our research area using a benchmark.

- The last part concludes the work with Chapter 11.

Part I

Background

2. Dynamic Analysis

2.1 Introduction

In this chapter we introduce dynamic analysis in Section 2.2, explain how it evolved in Section 2.3, and compare it to static analysis in Section 2.4. In Section 2.5, we give an overview of the notations used for describing system properties. Subsequently, in Section 2.6 we explain the use of automatic instrumentation in the context of runtime verification. Some algorithms used for runtime verification are introduced in Section 2.7. The issues related to overhead brought about by runtime verification are discussed in Section 2.8 while Section 2.9 concludes the chapter.

2.2 What is Dynamic Analysis?

Dynamic analysis is the process of verifying a system (referred to as the target system) while it is executing. In more technical terms, dynamic analysis will refer to all the techniques used to verify the system's properties for one particular execution trace obtained by executing the program. [35]. To perform such an analysis, we would require two basic components over and above the target system: (i) a monitoring mechanism; (ii) a verification mechanism. Monitoring the system entails the elicitation of events while they are occurring. These events are then communicated to the verification mechanism which verifies that the sequence of events adheres to the target system's specification. If a violation is found, then, the verification system raises an alarm or possibly reacts in some way so as to revert the target system back to an acceptable state. The following subsections will give a more detailed account of the phases which dynamic analysis comprises.

2.2.1 The Phases of Dynamic Analysis

We will now give an outline of the various phases involved in different dynamic analysis approaches. Figure 2.1 shows a block diagram of the dynamic analysis architecture. The numbers in the figure correspond to the following list of five phases.

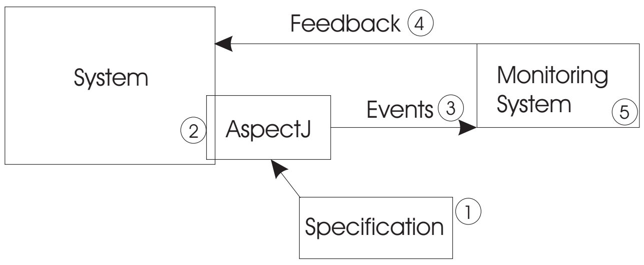


Figure 2.1: The phases of dynamic analysis.

1. **Specification.** The first phase includes the specification of the system's properties in some kind of formal notation. The choice of the notation depends on the domain of the problem.

2. **Instrumentation.** Once the system properties are specified, these must be instrumented into the system code. This process involves the generation of the monitoring code which concretely ensures that the specified properties are not violated. As soon as the system properties are in the same form as the system itself, they can then be instrumented into the target code. In some cases the monitoring code is not instrumented in the system code. Rather, a monitoring system is run in parallel to the monitored system and this will circumvent the need of the full instrumentation of monitoring code (but simply instrument the code which generates the events).

3. **Monitoring.** Subsequently, the actual monitoring of the system is carried out. This can either be done online or offline. This means that either the system is actually running and the monitoring mechanism is running synchronously or else a trace of the system's execution is saved and then it is verified at a later time.

4. **Handling violation of properties.** The next phase would be to raise exceptions on the detection of a property violation. Another possibility is that rather than simply raise an exception (as a notification of the violation), an automatic fault-handling mechanism may be implemented. This means that when a problem is detected an automatic action is carried out by the verification mechanism to correct the problem.

5. **Mitigating the impact of verification.** A final possible phase is the mitigation of the impact of verification. The purpose of this phase is to try to minimise or possibly eliminate the impact of the instrumented monitoring code on the actual system. It is not always possible to simply remove the monitoring code from the target system. In the first place, considering

the removal of the monitors assumes that the monitoring code is simply there for testing purposes, when in fact this is not always the case. Secondly, removing monitors from a system may bring about undesired effects which did not emerge due to the monitors. Two possible approaches to this problem would be to give guarantees (using metrics) on the effects of the added code, and/or allow the system to adapt itself (using reflection) at runtime to leave out certain code which checks for properties which may have been verified earlier in the execution.

2.3 Flavours of Dynamic Analysis

The following are various software design methodologies or architecture patterns which incorporate dynamic analysis to assure system properties. For each of these methodologies, we will explain which of the above phases it includes.

2.3.1 Design by Contract

Design by contract [72] has been developed with the aim of creating more secure software by manually inserting checks in the code which correspond to a *contract*. These checks, better known as assertions, are then verified during runtime. This approach is quite primitive when considering the amount of effort which is left to the developer to do. In design by contract, the system properties are inserted as preconditions or post-conditions in methods or as invariants for classes. Hence there is no need for code instrumentation [18], since the property specifications are manually inserted in the actual system implementation. It is important to note that there is no option of offline monitoring since the monitors become an integral part of the actual system. There is, however, the possibility of compiling a version of the program which simply ignores all the monitoring [72].

In the verification of security-critical systems, the aim is that the verification process is automated as much as possible so that the possibility of human error is eliminated. Therefore, design by contract has quite a considerable disadvantage when considering the amount of human intervention involved.

2.3.2 Runtime Verification

Runtime verification [25, 91, 67, 10] has also been developed to verify software against a formal specification. However, it differs from design by contract in various aspects. First of all, the specification of properties is done through formal notation and it is usually automatically instrumented into the target system. Runtime verification not only ensures that no system properties are violated, but it also provides the mechanism to correct the behaviour so that the system can continue to function. Therefore, the extra code for runtime verification includes both the monitoring code (which extracts the sequence of events (trace) from the current program execution) and the code which is triggered upon a violation of the properties.

If the specification of security properties is as cumbersome as writing the program itself, there is probably the same chance of making mistakes. Hence runtime verification is much more appropriate for security-critical systems because it uses formal notation for specifying security properties. The advantage is that usually, formal notation is very succinct and much more abstract than the actual implementation.

Although strictly speaking runtime verification should occur during the runtime of the target system, variations of runtime verification allow for both online and offline verification of the execution trace. This is motivated by the fact that the overhead introduced by the runtime verification code is sometimes too large. The alternative is to limit the overhead to the extraction of the trace from the running program (similar to creating a log) and subsequently the actual verification of the trace is performed later on.

2.3.3 Monitoring-Oriented Programming

Monitoring-oriented programming is a paradigm which combines the specification and the implementation of a system [19, 18]. It goes further than runtime verification in that it not only specifies properties to detect violations and raise exceptions, but the violation handling mechanism is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. For example, consider a system which does user authentication using a security policy in monitoring-oriented programming. Monitoring-oriented programming is a very flexible architecture and allows for the monitoring code to be separated from the actual system code. It also allows the monitoring code to be run asynchronously with respect to the system being monitored.

2.3.4 Runtime Reflection

Runtime reflection [68] adds yet another idea over and above monitoring-oriented programming methodology: it incorporates a diagnosis layer just above the monitoring layer. The purpose of this layer is to identify the type of failure that occurred rather than simply detect that a failure has occurred. This enables the system to give an *explanation of the current system state* [68]. One advantage of this extra layer is that it is more easily applicable in a distributed system where distributed parts send their monitoring information but the diagnosis is carried out at a central system. Furthermore, there is more separation of concerns in that the monitoring system simply obtains values representing the state of the system, while it is up to the diagnosis layer to interpret the system state. Subsequently, the mitigation layer performs an operation in response to that interpretation. It is important to note that the runtime reflection architectural pattern can still be achieved by using an monitoring-oriented programming methodology by implementing the diagnosis layer in the monitor-triggered code.

The idea of having an *explanation* for ending in a bad state is very desirable. This is more especially so when dynamic analysis is used during the testing phase to identify errors, since this would be of great help to the developers.

2.3.5 Online vs. Offline Verification

Recall that dynamic analysis is used to verify the system's properties for one particular trace of events during the actual execution of the system. The monitors are responsible to record a trace of events which are eventually analysed. The analysis of an execution trace can be done online or offline [18]. Online verification is when the target system and the verifying system are run synchronously in parallel (either as two separate systems or as a single instrumented system). The advantage of such a configuration is that the verification system can correct the detected property violations during the execution of the system. However, this poses more challenges for the verification process since the trace is not available as a whole, but becomes available progressively in synchrony with the actual execution. The fact that the trace is not all available means that certain efficient algorithms which are able to verify the trace backwards or as a whole [82, 80] cannot be used. However, there still exist efficient algorithms which are able to work on-the-fly [55, 56]. The main techniques used for runtime verification are dynamic programming [55] and term rewriting [23, 54, 53, 56].

Choosing between online and offline verification, one should consider whether the main purpose of the runtime verification is to find errors — during testing — or to find and automatically correct errors — after deployment. In the first case, it is practically irrelevant whether the verification is performed online or offline. Hence, it is preferable to use offline verification in such a circumstance. At the same time, offline verification reduces the overhead of verification since this is postponed to a later time and need not be done while the actual system is running. It is important to note that we still need the monitors to report the relevant events which are taking place.

2.4 Static vs. Dynamic Analysis

An important difference in software analysis is whether this is done before or during runtime. Static analysis will be used to refer to all the techniques (including theorem provers and model checkers) used to verify a program for all possible execution paths before the actual execution. On the other hand, the term dynamic analysis is usually used to refer to all the techniques used to verify the system's properties for one particular execution trace obtained by executing the program. Being able to verify properties for all possible executions paths [88, 91], makes static analysis a very desirable objective. However, for the algorithm to be tractable when applied to systems of a practical magnitude, static analysis depends on having a decidable domain. Various abstraction and reduction techniques have been proposed to scale up static analysis, but full verification of large-scale software systems is still largely unattainable [44, 91, 37]. In contrast, using dynamic analysis, one checks that a given system property holds along a particular execution path [91]. This is particularly useful to ensure that at no time during the execution of the system, are any of the system properties violated. Conversely, it can also identify execution paths along which the properties to be verified are not satisfied [91]. Essentially, dynamic analysis links the abstract specification to the actual concrete implementation [66, 85]. Thus, dynamic analysis can be used as a protection from potential faults at runtime, by implementing monitors to react to any property violations encountered [44]. Another motivation for using dynamic analysis is that certain information is only available at runtime. Furthermore, behaviours of the system may possibly depend on the environment where the system is running [25].

Although, there is this fundamental difference between static analysis and dynamic analysis, ways have been proposed in which these two approaches can complement each other by exploiting the benefits of one to aid the other. Ernst [35], in fact, argues that the difference between static analysis and dynamic analysis is over-emphasized. Complementarity can be achieved by applying both approaches and taking advantage of the "soundness" of static analysis while also benefiting of the "efficiency and precision" of dynamic analysis [35]. Soundness refers to the fact that any statically verified property holds for any possible path. Dynamic analysis is efficient because it needs to verify only a single path and precise because the properties are verified on the actual implementation and not on an abstracted version of the system. A similar idea is used by Zee et al. [91] by incorporating a runtime checker on top of the Jahob static verification system [91]. The purpose of this integration, is to help the developers in identifying errors in the system by showing concrete executions where the errors arise. Colcombet and Fradet [24], use static analysis to avoid unnecessary runtime checking. A similar approach is used by Aktug [2]. Havelund [51], on the other hand, suggests the use of runtime verification to guide the model checker and thus reducing the search space.

Another way in which static analysis and dynamic analysis can complement each other is for test-case generation. Static analysis can be used to "intelligently" generate test cases for a dynamic analysis tool to find errors during testing [8]. This is possible because static analysis provides the structure of the program and thus it is easier to produce test cases which cover most of the program paths.

2.5 Logics and Automata for Dynamic Analysis

In the coming chapters, we will consider and use various languages and logics which have been used for the specification of system properties. To facilitate their introduction later on, in this section we will provide a classification for these logics.

2.5.1 Models of Time

Each temporal logic is based on a particular model of time. The first classification is whether or not we specify properties with real-time quantities. Without real-time quantities we can simply represent ordered sequences of events. Furthermore, real-time quantities can either be integer or real values. The second classification is whether we consider time in continuous real-time or simply as discrete points. Finally, a third classification is whether we consider time to be linear or branching. Each of these combinations provide different expressive power and therefore different computation complexities for their verification. What follows is a discussion of these classifications with their respective characteristics, advantages and disadvantages.

Dense, Discrete Real-Time and Non-Real-Time Models of Time

Dense real-time means that time can be specified in real-numbers and hence we can refer to any particular moment in time. This is arguably the model closest to the physical world, which operates in continuous time. For example, timed automata [4] are based on this dense-time model. This provides a powerful model which allows us to specify features such as liveness, fairness, non-determinism, periodicity, bounded response and timing delays [4]. However, such a model poses a number of challenges as regards to the computability and complexity of the properties that we would like to enforce. The good news is that there are ways to bypass this problem. For example, region automata [4] are used to mimic the actual timed automata. Other literature [57, 16] propose digitisation to solve the problem of handling the dense-time model. This would result in the digitisation of the specified system and properties, thus transforming their underlying model of time into a discrete model.

The discrete model of time is very commonly used. Metric temporal logic [6], PSL [32] and QDDC [75] are all examples of logics based on this model. It is much more convenient to verify real-time properties which are based on natural numbers rather than real numbers. For example in [47], Halbwachs et al. suggest the use of the language Lustre to verify real-time systems by issuing an event for every second. Obviously, this can only be done for a discrete time model. A similar approach has been suggested by Gonnord et al. [45]. Although using the discrete time model seems to be limiting expressivity, many practical problems involving real-time can still be effectively verified [57].

We can further abstract the notion of time and limit ourselves to time-independent trace properties. Such logics (for example, linear temporal logic) have been extensively studied and very efficient algorithms have been proposed [55, 37, 42]. For many practical applications this model offers all the necessary expressivity. Hence, whenever we can limit ourselves to this time model, we will have the benefit of more efficient algorithms.

Interval vs. Non-interval Time Models

Properties can be specified either over points in time or over a set of points in time — an interval. Consider the following example: “Within any one hour period hour there should never be more than three bad logins” is a property specified on a time interval. Clearly, certain temporal properties are much more easily specified in this manner. Specifying the same property without the concept of an interval would be something like: “At no point should the count of bad logins, starting from the point which is one hour before (the one being considered), exceed three”. The idea of intervals has been proposed by various authors [90, 71]. The advantage of looking at intervals rather than points in time stems from the fact that there is no distinction between past and future. This is very important in runtime verification since the “future” is not available.

However, the notion of intervals in system properties, introduces new complexities for verification. This is because when we specify a property on an interval, the number of sub-intervals in that interval is equal to the number of all possible subsets of time points in that interval. Therefore, if we take a naïve approach in verification, most properties specified on intervals in dense time (such as interval duration logic) are undecidable [76]. However, this problem can be overcome by using particular subsets [76] or some kind of conversion such as digitisation [16].

Linear vs. Branching Time

Practically in every program at any point there may be multiple possible actions each leading to different traces. The behaviour is more like a tree than a linear chain. However, it sometimes suffices to consider only one of these branches, i.e. without considering all the other possible branches. Therefore, a classification of temporal logics has emerged. This is the distinction between linear and branching temporal logics [65]. In the linear model, a trace of the program is considered as a linear sequence of states where each state has one possible subsequent state. In contrast, in the branching model, at each point in time, all the possible execution paths are considered and thus a computational tree can be generated (where each node may have various possible successors). This distinction is however more important in static analysis rather than in dynamic analysis since we can only consider one execution trace (i.e. the one running at the time of verification).

2.5.2 Classifications of Logics

The literature is full of proposed formal notations for specifying system properties. Different authors have suggested different specification notations according to the particular domain, trying to improve one aspect or another of the existing notations. An important trade-off that should be highlighted is that the greater the expressivity of the formal notation used, the more complex is the algorithm for its verification. Some systems also propose their own specific language. Other systems which have been proposed go beyond by actually providing meta-languages which allow the user to specify other logics. For example, in the case of Eagle [10] and Maude [22], the proposed architecture provides the basic temporal logic constructs and then allows the user to create his own domain-specific logic.

In the following two subsections, we describe two classifications of logics.

Infinite and Finite Trace Logics

An important issue that arises with dynamic analysis is that the available trace which needs to be verified is not a complete one. In other words, it is still being created during the execution. Therefore, certain algorithms which work on infinite traces cannot be used without special adjustments. For example, when using Büchi automata (whose acceptance condition relies on infinite repetition of a set of states which include a final state), one such possible adjustment is given by Giannakopoulou and Havelund [43].

A number of temporal logics define their properties on infinite paths. A problem arises with liveness properties. A liveness property is a property which states that *eventually* something should happen. Therefore, it is difficult to verify such a property on traces which are being verified while they are generated. Put differently, while verifying we only have available a finite prefix of a possibly infinite trace. Therefore, if an eventuality has not occurred yet, we cannot say that the property does not hold on the actual trace (since we only know its prefix). To tackle this issue various alternatives have been proposed. Some logics and verification systems introduce new concepts to represent the “maybe” (or “not yet known”) state at which we cannot yet decide whether a property holds on a given (incomplete) path [11, 33, 30]. For example, some authors [33, 32] provide extra operators over and above the standard temporal logic to include a strong and a weak version of each operator. In the weak version, a liveness property holds if the eventuality has not yet occurred (in a finite trace) while in the strong version it does not. Similarly, a four-valued semantics [11] is defined which rather than simply returns true or false, can also return *possibly true* or *possibly false*. Furthermore, Finkbeiner and Sipma [37] proposed the use of statistics to be able to give the actual number of times that eventualities were fulfilled.

Future Time vs. Past Time Logics

Using a temporal logic we can specify properties such as “in the *next* state, the alarm should sound”. This property is specifying a constraint on the future. Similarly, we can specify properties on the past such as “in the *previous* time unit, we should have received a request”. Properties which can be expressed on the past can also be expressed on the future. In fact, it has been shown that past time LTL and future time LTL have the same expressive power [40]. However, one should note that past time temporal logic is more succinct than future time temporal logic [70]. An interesting approach presented by Gabbay [39], is to integrate the past and the future in a complementary way. This is achieved by using the past temporal logic to express what should have happened in the past and the future temporal logic to express what should be done in the future. For example, this can be used to write properties of the form "if A was true, then do B".

In runtime verification, separating the past and the future may cause some problems. Most notable is the problem of inconclusive results on future properties. For example, when verifying the property that *eventually A* will occur, and *A* has not yet occurred in the current execution, one cannot conclude whether the property is satisfied or violated. This problem can be solved by eliminating the distinction between past and future using the concept of intervals. In fact, this is an important advantage of interval logics which brings complete symmetry between past and future [90].

2.5.3 Automata in Verification

Automata have been extensively used in verification [29, 42, 26, 14, 31, 86, 37, 38, 85, 15]; especially for efficiently deciding whether a property is violated at a particular state. A very attractive advantage of using automata is that they are a pictorial representation. Since we intend to make our architecture "easily" usable for developers this representation may be more intuitive than other textual representations of the security properties.

2.6 Instrumentation Approaches

The monitoring and verifying code requires that, somehow, it is instrumented with the code of the system being verified. There are various ways of achieving this. In Temporal Rover [30] the code is inserted in the actual system code as if the logic is part of the system. This can be considered as though the instrumentation is being done manually. In other cases, such as in MaC [67] and PathExplorer [82], the instrumentation has a higher level of automation since the specification is separated from the actual system code. We do not want to leave the instrumentation to be done manually, because this is error-prone and we do no want our error checking mechanism to be error-prone! Therefore, it is desirable to find a way to automate the instrumentation of the monitoring code [25].

Aspect-Oriented Programming used for Runtime Verification.

Aspect-oriented programming allows the user to specify particular points in the system code where a piece of code will be automatically inserted. The concept of using this technique in runtime verification as a means of instrumentation is far from new [74, 27, 86, 14, 60, 41, 24, 38]. However, there are various levels and ways in which aspect-oriented programming can be used. For example using AspectJ [61], the user can write the monitoring code in a separate aspect, leaving the code of the actual system clean from monitoring code and at the same time, the code regarding the monitoring is all concentrated in one place. As an example, consider the scenario where we want to stop (or warn) users of a code library in the case of wrong usage. The rule which should be verified is that the library should be initialised before any other method is used. The code which blocks any method call before initialisation is shown below:

```
Object around():(execution (* Library.*(..)) && !execution(*
Library.initialization(..))) {
  if (initialized)
     return proceed();
  else
    return "LIBRARY: Library must be initialized.";
}
```

Using the “*” wildcard, we have managed to check for initialisation before the execution of any possible method apart from the one whose name is *initialization*. One should note the efficiency of implementing such logic in a few line of code rather than inserting a condition at the start of all the methods in the library. Furthermore, adding further methods to the library does not necessitate any modifications to the code handling the property.

Having automatic code injection and all monitoring code in one aspect is much less error-prone, but sill requires a lot from the developer and thus, there is still a lot of possibility for error. Hence, various authors [27, 14, 86, 74] have suggested the creation of an automatic way to generate AspectJ directly from the system's specification.

A slightly different approach has been proposed [24] where the specifications are directly weaved into the graph representing the program without going to the intermediate stage of creating aspects. Similarly, Fradet and Hong Tuan Ha [38] have suggested the specification of the properties in a special language semantically equivalent to timed automata and the program to be verified is also abstracted into timed automata. Subsequently, the weaving is performed on timed automata. The advantage of this approach is that the weaving is visible to the programmer. Furthermore, in specifying the properties the programmer can focus on the semantics of the properties rather than the syntax as in the case with using an aspect-oriented programming language such as AspectJ.

2.7 Algorithms for Verification

Verification usually consists in executing the automaton (representing the property) in parallel with the system being verified. If the automaton reaches an undesirable state, then the system has reached a bad state.

In particular cases it is not possible to have the complete automaton from the beginning; either because the automaton is infinite, or because it is too large to fit in memory. This is the case with symbolic automata — which may be infinite if explicitly unfolded. The reason is that symbolic automata allow the use of variables which can take an infinite number of values. It is sometimes necessary for automata to be generated on-the-fly. Therefore, it is imperative that whenever we use such automata for verification, we use an on-the-fly verification algorithm. In other cases, the conversion of the formula into automata may not be trivial or there may exist straightforward algorithms which can be applied directly on the formula.

On-the-fly Algorithms

Another subset of algorithms are those which divide the predicates to be verified into two parts. The basic idea is that since we do not have the whole trace available during runtime verification, we must satisfy the formulas as we progress through the execution. Hence, we split the formula into two parts: the part which we should check "now" and the part which we will check later; hence the name "on-the-fly". For example, verifying a formula having the $\Box$ (*always* operator) will entail the verification that the formula holds at the current state and also that it will hold in the next state. In other words, checking that p always holds requires that p currently holds and that in one step p always holds. Similarly, a formula having the $\Diamond$ (*eventually* operator) will require a test of whether or not the formula holds at the current state and, if it does not, it has to be checked again in the next state. Such algorithms exploit the recursive nature of the temporal logic and use the results from the consecutive states to reach the result of the current state. Using automata to represent formulas, we will in fact be generating the verifying automaton on-the-fly according to the previous state.

The idea behind dynamic programming is the very same idea of reaching a result based on the previous results. In fact, dynamic programming has been suggested as a very efficient way of checking whether or not a formula holds at a certain state of the execution [82]. This algorithm exploits the fact that the properties can be recursively defined and hence, there is no need to test the whole trace at each point in time; it is sufficient to consider the current state while having the computed result of the rest of the trace. Thus, this gives a time complexity of $O(n)$ The main drawback is that the trace has to be traversed backwards and hence cannot be used to monitor a trace during its execution. However, later on, the algorithm was improved to perform verification at runtime [55] and thus this drawback was overcome.

Another suggested similar approach is term rewriting [23, 54, 53, 56]. In this case, a formula is transformed (reduced) after each event (hence the word rewriting) until it is satisfied or violated at a later state in the trace. Very similar work has also been done [10] where the temporal formula can be separated into three parts — past, present and future — connected with boolean connectives.

2.8 Overheads in Using Dynamic Analysis

Since dynamic analysis involves some kind of monitoring and extra verification, then it obviously introduces an overhead to the system being monitored. Such an overhead can have many negative consequences on the system which must react under real-time constraints.

The kind of optimisations which are possible are very specific to the architecture used and at which phase of the verification process this is carried out. For example when using automata, there is the concept of pruning [37] or collapse [31].

Other proposed optimisations try to avoid unnecessary checking during runtime [38, 89]. Colcombet and Fradet [24] propose a mixture of static and dynamic analysis so that properties which can be verified upon inspecting the source code are not unnecessarily tested-for during runtime. Thane [87] classifies various types of monitors and for each type explains how it can or cannot be removed from the target system. Havelund and Roşu [56] suggest to optimise boolean functions and evaluate the predicates such that probabilistically they would have the minimum runtime cost.

Drusinsky [31] analyses the growth of p-trees under different restrictions. In this way the upper-bound of the size of the tree (and hence memory usage) can be guaranteed. To save memory usage, Courcoubetis et al. [26] propose the use of hashing without collision detection to store program state with the disadvantage of possibly missing some states.

In case studies carried out by Bhargavan et al. [13], the overhead of runtime verification is quite large. Hence, two kinds of optimisations (abstractions) were proposed in the scenario of a packet routing algorithm. One is that the verification was limited to a certain number of nodes rather than all the nodes (population abstraction) while the other limited the verification to a particular type of packets (packet-type abstraction). However, these optimisations are reasonable if the aim of the verification is that of finding errors in the testing phase instead of ongoing monitoring of the target system.

In general, runtime verification overheads cannot be completely removed. Instead, the above approaches reduce the overheads to an acceptable level depending on the application.

2.9 Conclusion

Runtime verification has evolved as a compromise to the scalability problem of static analysis and the lack of coverage of testing. The initial idea of design by contract has been further developed so that monitoring code is automatically instrumented with the target system. One approach to automatic instrumentation is aspect-oriented programming. The spectrum of runtime verification — be it in notations used and be it in algorithms employed — is very wide. The reason is that each proposed approach is more targeted to particular applications. Runtime verification introduces an overhead on the monitored system. This may be detrimental especially in the case of real-time systems. A lot of work still needs to be done to analyse the magnitude and effect of the overhead of monitoring over the target system.

3. Real-Time Logics

3.1 Introduction

This chapter introduces a number of real-time logics used in this work. Duration calculus is a formal approach to designing real-time systems. Its syntax and semantics are introduced in Section 3.2. In the next section we introduce counterexample traces which are an implementable subset of duration calculus. QDDC, another subset of duration calculus, is introduced in Section 3.4. Phase event automata which can be used to monitor counterexample traces are introduced in Section 3.5 while Section 3.7 concludes the chapter.

3.2 Duration Calculus

Duration calculus [90, 49] formalises the concept of duration formulae — properties over intervals of time, which are based on the value of the underlying boolean states (variables which change over time). The calculus is based on two main concepts: integration — to measure for how long a boolean variable holds over an interval — and the chop operator — which splits an interval into two with different formulae which must hold on the two subintervals. Various other operators can be defined in terms of these basic ones together with the boolean connectives, to enable the expression of properties such as:

$$\Box(\int badstate > 3mins \Rightarrow l > 60mins)$$

This can be read as "for every subinterval, if the system is in a bad state for more than three minutes, then the length of that subinterval must be greater than an hour. In other words a system cannot be in a *badstate* for more than three minutes in any interval whose length is smaller or equal to an hour".

In this case, *badstate* is a state variable which changes between true and false as time goes by. Thus, a state variable can be considered as a wire in an electrical circuit which can have a low voltage or a high voltage. It can also be considered as a function of time which, given a particular time, returns a boolean value. By using *1* and *0* to represent *true* and *false*, the mathematical integral can be applied on a variable. This would yield the total time that the variable was true. The term *true* (and *false*) when used as state variables are functions which return true (or false) all the time. Thus, the integral of *true* returns the length of the interval while the integral of *false* is always zero. The set of state variables is represented by $SVar$ while state variables are represented by the letters *P, Q, R,* $\ldots$.

By applying the integral, a real value is obtained from a boolean variable. This opens up the possibility of using real operators such as $+$ and $-$, and comparison operators such as $<$ and $>$. Thus, in a duration calculus formula, there are both real functions and boolean functions with real-valued arguments. The set of real functions is represented by *FSymb* while the set of boolean functions is represented by *RSymb*. The elements of *FSymb* are referred to by $f^n, g^m, \ldots$ where n and m are the number of arguments. Similarly, the elements of *RSymb* are referred to by $p^n, q^m, \ldots$. Real-valued arguments are not restricted to the integral function of state variables. Rather, time-independent real-valued variables are very useful, for example, to compare the integral of a state variable. The set of real-valued variables is represented by $GVar$ and the letters *x, y, z,* $\ldots$ are used to refer to its elements.

The following list summarises the symbols explained above:

0 False.

1 True.

$\int$ The integration operator which measures the duration a state expression evaluates to true.

SVar The set of boolean-valued state variables P, Q, R,..., whose values depend on time.

GVar The set of real-valued variables x, y, z,..., whose values are time-independent.

FSymb The set of real functions f^n, g^m,..., $(n, m \geq 0)$ where f^n takes n real-valued arguments and returns a real value. For our purposes, the functions are assumed to be $+$ and $-$.

RSymb The set of boolean functions p^n, q^m,..., $(n, m \geq 0)$ where f^n takes n real-valued arguments and returns a boolean value. For our purposes, the relations are assumed to be $<$, $\leq$, $=$, $\geq$ and $>$.

Using state variables as the bases, and the usual binary operators, state expressions can be constructed. As before, the integral can be applied to yield the time during which the expression was true. The set state expressions is defined inductively as follows:

$$\mathit{StateExpr} ::= 0 \mid 1 \mid SVar \mid \neg\ \mathit{StateExpr} \mid \mathit{StateExpr} \vee \mathit{StateExpr}$$

A term in duration calculus is always real-valued — it can be an integral of a state expression, a real-valued variable, or a real-valued function. This is defined as follows:

$$\mathit{Term} ::= \textstyle\int \mathit{StateExpr} \mid \mathit{GVar} \mid f(\mathit{Term}, \ldots, \mathit{Term})$$

Using the terms as building blocks, a duration calculus formula is a boolean function which takes an interval as input. A duration formula is formed by using boolean functions on real-valued terms and applying the boolean operators on them. Note that the boolean symbols $\vee$ and $\neg$ are overloaded — used for state expressions and formulae. An additional operator is the chop operator. The chop operator divides an interval into two subintervals and applies the left formula on the first subinterval and the right formula on the second subinterval. The set of duration calculus formulae is defined inductively as follows:

$$\begin{array}{rcl} \mathit{Formula} & ::= & p(\mathit{Term}, \ldots, \mathit{Term}) \mid \neg\, \mathit{Formula} \\ & \mid & \mathit{Formula} \vee \mathit{Formula} \mid \mathit{Formula} \frown \mathit{Formula} \end{array}$$

3.2.1 Semantics

In duration calculus, time is modelled as the set of non-negative real numbers:

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

A duration calculus formula is a composition of boolean expressions and duration calculus operators specified over an interval. We define an interval as follows:

$$\mathit{Interval} \stackrel{\text{def}}{=} [b, e] \ \mathit{where}\ b, e \in \mathbb{T} \wedge b \leq e$$

A boolean state is a boolean variable which may change over time. Based on boolean states, one can define boolean expressions using standard boolean operators ($\vee$, $\neg$). As usual the abbreviations $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ can be used. The truth and falsity of a boolean state depends on the interpretation being considered. An interpretation I can be seen as a function which returns the temporal behaviour of a given boolean state:

$$I \in \mathit{BooleanState} \rightarrow (\mathbb{T} \rightarrow \mathbb{B})$$

An interpretation can be lifted to give the value of a boolean expression by applying the interpretation on the constituent boolean states. In other words, $I(X \vee \neg\, Y)(t)$ is defined as $I(X)(t)$ *or* $\neg\, (I(Y)(t))$.

It is assumed that the boolean variables obey the finite variability property — over any finite interval, a boolean state may only have a finite number of discontinuous points.

The truth of a formula thus depends on the underlying interpretation of the boolean states. Under a given interpretation I, we can define what it means for a duration formula D to be true for a given interval $[b, e]$, which we write as $I \models_{[b,e]} D$.

The duration formula $\int P = n$ holds if boolean expression P holds for a total of n time units over the interval:

$$I \models_{[b,e]} \int P = n \stackrel{\text{def}}{=} \int_b^e I(P)(t)dt = n$$

The boolean operators can also be lifted over duration formulae:

$$\begin{array}{lcl} I \models_{[b,e]} \neg D & \stackrel{\text{def}}{=} & I \not\models_{[b,e]} D \\ I \models_{[b,e]} D \wedge E & \stackrel{\text{def}}{=} & I \models_{[b,e]} D \text{ and } I \models_{[b,e]} E \\ I \models_{[b,e]} D \vee E & \stackrel{\text{def}}{=} & I \models_{[b,e]} D \text{ or } I \models_{[b,e]} E \end{array}$$

The chop operator is used to split an interval into two subintervals where each subinterval satisfies the respective formula:

$$I \models_{[b,e]} D \frown E \stackrel{\text{def}}{=} I \models_{[b,m]} D \; and \; I \models_{[m,e]} E \; for \; some \; m \in [b, e]$$

Based on these operators, one can define other useful operators syntactically. The length (ℓ) of an interval can be measured and defined as follows:

$$\ell \stackrel{\text{def}}{=} \int 1$$

where 1 is the boolean state constantly true.

One can define other comparison operators on the duration of boolean expressions syntactically. For instance, $\int P \geq n$ can be written as $\int P = n \frown \text{true}$. Using boolean operators, the other comparators can also be defined.

The duration formula stating that boolean expression P holds (almost) everywhere throughout the given interval, written $\lceil P \rceil$, is defined as:

$$\lceil P \rceil \stackrel{\text{def}}{=} \int P = \ell \wedge \ell > 0$$

Based on the chop operator one can define what it means for a duration formula D to hold over some subinterval, written as $\Diamond D$, and defined as:

$$\Diamond D \stackrel{\text{def}}{=} \text{true} \frown D \frown \text{true}$$

The dual operator — stating that a duration formula D holds over all subintervals, written as $\Box D$, and can be defined as:

$$\Box D \stackrel{\text{def}}{=} \neg \Diamond \neg D$$

A duration formula is valid under an interpretation if it holds for all time prefixes:

$$I \models D \stackrel{\text{def}}{=} \forall t : \mathbb{T} \cdot I \models_{[0,t]} D$$

Finally, a duration formula is said to be a tautology if it holds under all interpretations.

$$\models D \stackrel{\text{def}}{=} \forall I \cdot I \models D$$

3.3 Counterexample Traces

In general, duration calculus is too expressive to be monitored with a bounded number of clocks. Consider the example given by Bouajjani et al. [1]:

$$\Box(\uparrow \pi \wedge (\ell > 1) \Rightarrow (\lceil \mathit{true} \rceil \wedge \ell = 1) \frown \lceil \neg \pi \rceil \frown \mathit{true})$$

where $\uparrow \pi$ is satisfied by an interval $[b, e]$, if π turns to true exactly at beginning at b:

$$\begin{array}{rcl} I \models_{[b,e]} \uparrow \pi & \stackrel{\text{def}}{=} & \exists \epsilon > 0 \cdot \forall t \in [b, b+\epsilon] \cdot I(\pi)(t) = 1, \text{ and} \\ & & \text{either } b = 0 \text{ or } \forall t \in [b, b-\epsilon] \cdot I(\pi)(t) = 0 \end{array}$$

The formula in the example states that exactly one time unit after a rising edge of π, it should turn to false. The problem is that in one time unit there may be an unbounded number of rising edges of π. One would need an unbounded number of clocks to verify such a property.

A class of useful implementable duration calculus formulae are known as the class of *implementables* [78, 84]. Ravn [78] shows how duration calculus can be expressed as implementables. The class implementables can be represented by another subset of duration calculus called counterexample traces [58]. Each counterexample trace has a corresponding phase event automaton which can be used to monitor the satisfaction of the counterexample trace [58]. The basic operator of the implementables is the *followed-by* operator $F \longrightarrow \lceil P \rceil$, stating that an interval which satisfies F should be followed by an interval where P holds. Introducing time limits on the length of F makes this operator very useful in practical applications. For example, $F \xrightarrow{\leq t} \lceil P \rceil$ defines the property that following any interval satisfying F, whose length is less than or equal to t, P should be true. This can be defined using duration calculus as follows:

$$F \xrightarrow{\leq t} \lceil P \rceil \stackrel{\text{def}}{=} \neg \Diamond(F \wedge \ell \leq t \frown \lceil \neg P \rceil)$$

A complete set of implementables is given by Schenke and Olderog [84].

3.3.1 Preliminaries

Before we give the syntax of counterexample traces, we must first introduce some abbreviations used by Hoenicke [58] — originally suggested by Chaochen and Hansen [17]. The formulae $\nwarrow S$ and $\nearrow S$ mean that a state expression is satisfied before (or after, respectively) a given point in time:

$$I_{[b,e]} \models \nwarrow S \overset{\text{def}}{=} b = e \land \exists\, m : \mathbb{R} \ \cdot \ m < b \land I_{[m,b]} \models \lceil S \rceil$$

$$I_{[b,e]} \models \nearrow S \overset{\text{def}}{=} b = e \land \exists\, m : \mathbb{R} \ \cdot \ m > e \land I_{[e,m]} \models \lceil S \rceil$$

More informally, $\nwarrow S$ holds on a point interval p such that an interval i exists which satisfies $\lceil S \rceil$ and where p is the end point of i. Similarly, $\nearrow S$ holds on a point interval p such that an interval i exists which satisfies $\lceil S \rceil$ and where p is the first point of i. We can further define $\updownarrow S$ which signifies that a change occurs in the value of S:

$$\updownarrow S \overset{\text{def}}{=} (\nwarrow \neg S \land \nearrow S) \lor (\nwarrow S \land \nearrow \neg S)$$

Conversely, to denote the fact that variable S does not change:

$$\not\updownarrow S \overset{\text{def}}{=} \neg \updownarrow S \land \ell = 0$$

Finally, to denote the fact that a variable does not change within an interval:

$$\boxminus S \overset{\text{def}}{=} \neg\, (\ell > 0 \frown \updownarrow S \frown \ell > 0)$$

Note that a variable change is allowed anywhere except at the start and end of the interval.

3.3.2 Syntax

In short, a *counterexample trace* is a sequence of phases with possible events in between. The sequence is represented by a *chop* operator ($\frown$) between each phase. An *event* is the occurrence of a change of a boolean variable from true to false or vice-versa. A *phase* is characterised by its predicate, which is a propositional formula on the boolean variables. This predicate must hold "almost everywhere" ($\lceil \mathit{Predicate} \rceil$) during the phase in which it is specified. The length of a phase can be restricted by using a lowerbound or upperbound on the length of the phase (ℓ). During a phase, we can prohibit any number of events from occurring using $\boxminus \mathit{NAME}$ where NAME is the event. In the following paragraphs we will expand more on counterexample traces and give their formal syntax.

Starting from the building blocks, the events which can come in between phases hold on point intervals as defined in the previous section. An event can either be a change in the state variable $NAME$ at a particular point, written as $\updownarrow NAME$, or a prohibition of a change in the state variable $NAME$ at that point, written as $\not\updownarrow NAME$. Both disjunction and conjunction can be used between events. A disjunction means that either one event or the other occurs, while a conjunction requires that both events occur simultaneously. Events are inductively defined as follows:

$$\begin{array}{rcl} event & ::= & \updownarrow NAME \mid \not\updownarrow NAME \\ & & event \vee event \mid event \wedge event \end{array}$$

The other type of building blocks are the phases. A phase can have three types of restrictions. The first involves a predicate which must hold throughout the interval. This is represented by $\lceil Predicate \rceil$. A predicate is any boolean expression on the state variables. It may also be *true* and thus, this will be satisfied by any interval whose length is greater than zero[1]. In the absence of a predicate, the phase will be defined by *true* which is satisfied by any interval, even if its length is zero. Another restriction which may be applied on a phase concerns its length. An upperbound $(<, \leq)$ or a lowerbound $(\geq, >)$ may be applied on the length of the interval but not both. Note that counterexample traces do not have the *equality* operator on ℓ. This in practise can be substituted by the other operators[2]. The third restriction on a phase is a list of events which cannot occur during that phase. This is represented by a conjunction of variables which are not allowed to change during the phase: $\boxminus NAME \wedge \ldots \wedge \boxminus NAME$. Phases are inductively defined as follows:

$$\begin{array}{rcl} \sim & ::= & \leq \mid < \mid > \mid \geq \\ phase & ::= & (true \mid \lceil Predicate \rceil)[\wedge \ \ell \sim t] \\ & & [\wedge \boxminus NAME \wedge \ldots \wedge \boxminus NAME] \end{array}$$

Finally, phases and events are connected together using the $\frown$ operator. Note that the first element of the chop must be a phase and the last element must be *true*. The reason for starting with a phase is that, by definition, events cannot occur at the start of an interval. The formula must end with *true* so that it is easy to identify when the whole formula is satisfied or not. The other elements of the chop are not restricted. As the name suggests, a counterexample trace is a trace which should not be satisfied. Thus, the chop of phases is negated to signify that satisfying the chop will in fact mean the violation of the represented

[1]Note that the definition of $\lceil - \rceil$ requires a non-zero length interval.

[2]For example consider the formula: $F \xrightarrow{t} \lceil P \rceil \overset{\text{def}}{=} \neg \diamond(F \wedge \ell = t \frown \lceil \neg P \rceil)$ can usually be replaced by: $F \xrightarrow{\geq t} \lceil P \rceil \overset{\text{def}}{=} \neg \diamond(F \wedge \ell \geq t \frown \lceil \neg P \rceil)$.

property. The syntax of counterexample traces is as follows:

$$\begin{aligned} \mathit{ce\_formula} \;::=\; & \neg\,(\mathit{phase} \frown (\mathit{phase} \mid \mathit{event}) \\ & \quad \frown \ldots \frown (\mathit{phase} \mid \mathit{event}) \frown \mathit{true}) \end{aligned}$$

3.3.3 Examples

To illustrate the meaning of counterexample traces, we will give a few examples:

Example 3.3.1.

$$\neg\,(\lceil A \rceil \wedge \ell < 1 \frown \lceil B \rceil \frown \mathit{true})$$

This means that an interval during which A is true for less than one time unit, followed by B being true, should never occur.

Example 3.3.2.

$$\neg\,(\mathit{true} \frown \lceil D \rceil \wedge \ell > 1000 \frown \lceil \neg\, \mathit{Alarm} \rceil \frown \mathit{true})$$

Now the formula starts and ends with *true*. For this reason, the formula is also disallowed for all the subintervals. Therefore, a subinterval during which D holds and whose length is greater than 1000 time units, can never be followed by a subinterval for which *Alarm* does not hold.

Example 3.3.3.

$$\neg\,(\mathit{true} \frown \updownarrow B \frown \lceil \mathit{true} \rceil \wedge \ell < 10 \frown \updownarrow B \frown \lceil \mathit{true} \rceil \wedge \ell < 500 \frown \updownarrow B \frown \mathit{true})$$

This means that for any sequence of three B events, the interval between the first two cannot be less than 10 time units while the interval between the second and the third B events cannot be less than 500 time units. For example, any three B events which occur in less than 510 time units will surely violate this formula.

3.3.4 Relationship to Duration Calculus

Counterexample traces are based on the dense-time model like duration calculus. However, with the restrictions placed on the syntax, counterexample traces are implementable unlike duration calculus. Most notably, counterexample traces do not have the integral on state variables, but only $\int 1$ which measures the whole duration of an interval. Another important issue is that the only negation allowed is the negation of the whole formula[3]. This is compulsory if the formula

[3]Note that this does not apply to boolean expressions.

needs to be converted into a phase event automaton. Boolean operators are also restricted; only logical *and* is allowed to connect phase restrictions. In spite of all these restrictions, counterexample traces have been shown to be at least expressive as the set of implementables [58].

3.4 QDDC

Another approach to have an implementable subset of duration calculus is Quantified Discrete Duration Calculus (QDDC) [75]. Although QDDC is based on a discrete-time model rather than dense-time, it is still very expressive.

Syntax of QDDC Formulae

Similar to duration calculus, the bases of QDDC are the state variables which vary between true and false with time. The difference is that in this case, the time is discrete and thus the value of the variables is defined at discrete points, not on continuous time. Let *Pvar* be the set of propositional variables. Let p, q range over propositional variables; P, Q over propositions; and D, D_1, D_2 over QDDC formulae. *True* and *false* are represented by *1* and *0* respectively. Parts of the syntax which are not relevant to our work are not included in this overview. The set of propositions *Prop* has the following syntax with the usual proposition logic operators:

$$P ::= 0 \mid 1 \mid p \mid P \wedge Q \mid \neg\, P$$

The integral equivalent in QDDC is the Σ. The integral returns a real value while Σ returns an integer value because of the different time domains. Similarly, η returns the length of the interval as an integer. Furthermore, because of the discrete nature of QDDC, it is possible to constrain the value of a state variable at a particular point. This is done by the operator $\lceil P \rceil^0$. To represent the fact that a variable is always true throughout an interval, the operator $\lceil\lceil P \rceil$ is used. Note that the first point of the interval is included while the last one is not. Furthermore, there is the *chop* operator which intuitively has the same meaning as in duration calculus. The rest of the operators are standard boolean operators. The syntax of QDDC is as follows:

$$D ::= \lceil P \rceil^0 \mid \lceil\lceil P \rceil \mid D_1 \frown D_2 \mid D_1 \wedge D_2 \mid \neg\, D \mid \eta\ op\ c \mid \Sigma P\ op\ c$$

where $op \in \{>, =\}$ and $c \in \mathbb{N}$

Semantics of QDDC Formulae

The behaviours of propositions and formulae are finite sequences of states ($\sigma = \sigma_0\sigma_1...\sigma_n$) where each state maps the proposition (or formula, respectively) to

$\{\mathit{true}, \mathit{false}\}$. The semantics of propositions are defined on a particular position in the behaviour such that σ_i represents position i in behaviour σ:

$$\sigma_i \models p \overset{\text{def}}{=} \sigma_i(p) = 1$$

An interval, $\sigma[b, e]$, is a sequence of states:

$$\sigma[b, e] = \sigma_b \sigma_{b+1} ... \sigma_e, \ \mathit{where}\ 0 \leq b \leq e \leq n$$

The semantics of formulae are defined on intervals:

$$\begin{array}{lcl}
\sigma[b,e] \models \lceil P \rceil^0 & \overset{\text{def}}{=} & b = e \ \mathit{and}\ \sigma_b \models P \\
\sigma[b,e] \models \lceil\lceil P \rceil & \overset{\text{def}}{=} & b < e \ \mathit{and}\ \forall\, i, b \leq i < e, \sigma_i \models P \\
\sigma[b,e] \models D_1 \frown D_2 & \overset{\text{def}}{=} & \exists\, i, b \leq i \leq e, \sigma[b,i] \models D_1 \ \mathit{and}\ \sigma[i,e] \models D_2 \\
\sigma[b,e] \models D_1 \wedge D_2 & \overset{\text{def}}{=} & \sigma[b,e] \models D_1 \ \mathit{and}\ \sigma[b,e] \models D_2 \\
\sigma[b,e] \models \neg\, D & \overset{\text{def}}{=} & \sigma[b,e] \not\models D \\
\sigma[b,e] \models \eta\ op\ c & \overset{\text{def}}{=} & (e - b)\ op\ c \\
\sigma[b,e] \models \Sigma P\ op\ c & \overset{\text{def}}{=} & \mathit{Card}\{i \mid b \leq i < e, \sigma_i \models P\}\ op\ c
\end{array}$$

An interval satisfies $\lceil P \rceil^0$ if its length is zero and satisfies P; an interval satisfies $\lceil\lceil P \rceil$ if its length is greater than zero and satisfies P throughout, except the last state; an interval satisfies $D_1 \frown D_2$ if it can be divided into two subintervals where the first satisfies D_1 and the second satisfies D_2; an interval satisfies $D_1 \wedge D_2$ if it satisfies both D_1 and D_2; an interval satisfies $\neg\, D$ if it does not satisfy D; an interval satisfies $\eta\ op\ c$ if its length satisfies the comparison $op\ c$; an interval satisfies $\Sigma P\ op\ c$ if the number of states which satisfy P, satisfies the comparison $op\ c$.

There are various other derived operators based on the above:

$$\begin{array}{lcl}
\lceil\lceil P \rceil\rceil & \overset{\text{def}}{=} & \lceil\lceil P \rceil \frown \lceil P \rceil^0 \\
\lceil\lceil P \rceil^+ & \overset{\text{def}}{=} & \lceil\lceil P \rceil \vee \lceil P \rceil^0 \\
\lceil\lceil P \rceil\rceil^+ & \overset{\text{def}}{=} & \lceil\lceil P \rceil\rceil \vee \lceil P \rceil^0 \\
\Diamond D & \overset{\text{def}}{=} & \mathit{true} \frown D \frown \mathit{true} \\
\Box D & \overset{\text{def}}{=} & \neg \Diamond \neg\, D
\end{array}$$

3.4.1 A Deterministic Fragment of QDDC

In this work we are particularly interested in a deterministic fragment of QDDC [45] which can be expressed as Lustre acceptors. The basic differences from the full QDDC are the following:

- an operator *begin*(P) which, given a variable P, returns true if the variable is true at the beginning of the interval;

- similarly an operator $end(P)$ which, given a variable P, returns true if the variable is true at the end of the interval;
- the operator $age(P)$ which returns the duration for which a given variable P has been continuously true up to the end of the interval;
- an operator G *then* F which works as a deterministic chop, i.e. starts to satisfy formula F as soon as G is no longer satisfied. Because of the *then* operator, any formula in fragment G which turns to false at a particular point of an interval, never evaluates to true later in that interval. This explains why the operators on Σ, η, and *age* are restricted to $\leq$.

The fragment is defined inductively as follows:

$$G ::= begin(P) \mid \lceil\lceil P \rceil \mid \eta \leq c \mid \Sigma P \leq c \mid age(P) \leq c \mid G_1 \wedge G_2 \mid G_1 \vee G_2$$

$$F ::= G \mid end(P) \mid G \; then \; F \mid F_1 \wedge F_2 \mid \neg \, F$$

The semantics are as follows:

$$\begin{array}{lcl} \sigma[b,e] \models begin(P) & \overset{\text{def}}{=} & \sigma_b \models P \\ \sigma[b,e] \models age(P) \leq c & \overset{\text{def}}{=} & e - n \leq c \\ \sigma[b,e] \models end(P) & \overset{\text{def}}{=} & \sigma_e \models P \\ \sigma[b,e] \models G \; then \; F & \overset{\text{def}}{=} & \exists \, m : \mathbb{N} \; \cdot \; b \leq m < e \mid \sigma_{[b,m]} \models G \wedge \sigma_{[b,m+1]} \not\models G \\ & & \wedge \; \sigma_{[m+1,e]} \models F \end{array}$$

where

$$n = \begin{cases} max\{i \mid b \leq i \leq e, \sigma_i \models \neg \, P\} & \text{if } \neg \, P \text{ occurred in [b,e]} \\ b - 1 & \text{otherwise} \end{cases}$$

The semantics of rest of the symbols remain the same as in duration calculus.

3.4.2 Relationship to Duration Calculus

Since QDDC is a subset of duration calculus, all of it can be translated to duration calculus. This translation is useful so that the properties proved on duration calculus can also be applied to QDDC. For example, the guarantees which will be given later on in this work regarding duration calculus can be directly applied to QDDC.

It is beyond the scope of this work to provide the complete translation with the necessary proofs. However, we could define a function $\mathcal{DC}$ accepts a QDDC formula and returns a duration calculus formula:

$$\begin{array}{rcl}
\mathcal{DC}(begin(P)) & \equiv & \nearrow P \frown true \\
\mathcal{DC}(\lceil\lceil P\rceil) & \equiv & \nearrow P \frown \lceil P\rceil \\
\mathcal{DC}(\eta \ op \ c) & \equiv & \int 1 \ op \ c
\end{array}$$

3.5 Phase Event Automata

Phase event automata [58] have been proposed for the purpose of amalgamating CSP, Object-Z and duration calculus. Their main advantage is that they can synchronize on events, handle data by using variables, represent real-time and provide parallel composition with simple compositional semantics.

3.5.1 Syntax

Phase event automata are expressive automata able to handle events, real-time and state-variables. Each location of the automaton has two invariants: one which constraints the state variables and another which constraints the clocks. On each transition, a guard constraints the state-variables and the clocks, and a number of clocks are reset.

A phase event automaton $\mathcal{P} = \langle P, V, A, C, E, s, I, E_0\rangle$ consists of the following components:

P : a set of *locations (phases).*

$V \subseteq NAME$: a set of typed *state variables.*

$A \subseteq NAME$: a set of boolean *event variables.*

C : a finite set of real-valued *clocks.*

$E \subseteq P \times \mathcal{L}(V \cup A \cup C \cup V') \times \mathbb{P}(C) \times P$: a set of *edges.* An element (p, g, X, p') represents an edge from p to p'. The current valuation of state and clock variables and events must satisfy the guard g. All clocks in X are reset after the transition.

s: $P \to \mathcal{L}(V)$: a labelling function that associates each location with a *state invariant* that must hold while this location is active.

I:$P \to \mathcal{L}^c(Clocks)$: a function assigning a clock invariant to each location.

$E_0 \subseteq \mathcal{L}(V) \times P$: a set of *initial edges.* An element (g, p) allows the automaton to start in p if the state predicate g holds.

3.5.2 Operational Semantics

A *configuration* of a phase event automaton is (p, Y, β, γ, t) where p is the location, Y is a set of events, β is a valuation of variables, γ is a valuation of clocks, and t is the duration spent in location p.

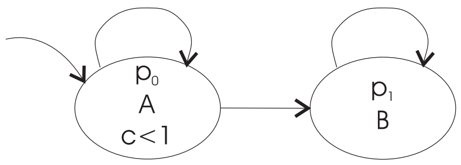


Figure 3.1: An example of phase event automaton.

A *run* is a sequence of configurations $(p_1, Y_1, \beta_1, \gamma_1, t_1), \ldots, (p_n, Y_n, \beta_n, \gamma_n, t_n)$ which satisfies the following conditions:

- $Y_1 = \emptyset$,
- $\beta_1 \vDash g$ for some initial edge $(g, p_1) \in E_0$,
- $\gamma_1(c) = 0$ for all $c \in C$,
- $t_i > 0$ for all $i \in 1..n$,
- $\beta_i \vDash s(p_i)$ for all $i \in 1..n$,
- $\gamma_i + t_i \vDash I(p_i)$ for all $i \in 1..n$, for all $i \in 1..(n-1)$ there is an edge (p_i, g, X, p_{i+1}) with $\beta_i, \beta'_{i+1}, \gamma_i + t_i, Y_{i+1} \vDash g$ and $\gamma_{i+1} = (\gamma_i + t_i)[X := 0]$.

This means that by $\beta_1 \vDash g$ the automaton starts from some initial edge such that the valuation of variables satisfies the guard on the initial edge (g, p_1) and enters location p_1. At this first instance no events can occur since, by definition, no events can occur at the starting/ending point of an interval; hence $Y_1 = \emptyset$. The clocks are initially all set to zero and therefore we require that $\gamma_1(c) = 0$ for all clocks. $t_i > 0$ is required to ensure that the automaton does not stay for zero time in any location. Each location has an invariant and a clock invariant and it must be ensured that while the automaton is in a location, these invariants are satisfied. This explains the need for $\beta_i \vDash s(p_i)$ and $\gamma_i + t_i \vDash I(p_i)$. Finally, each configuration should be related to the next through an appropriate edge (p_i, g, X, p_{i+1}) such that the valuations at p_i (β_i) and p_{i+1} (β'_{i+1}), the clock valuations $(\gamma_i + t_i)$, and the events (Y_{i+1}) satisfy the guard g. When an edge is taken, the clock valuations are updated $(\gamma_{i+1} = \gamma_i + t_i)$ and the clocks in set X are reset $([X := 0])$.

Example 3.5.1. Consider the simple example where the automaton accepts traces such that A should be true for the first time unit and then B is always true. This is shown in Figure 3.1.

3.6 Translating Duration Calculus to Phase Event Automata

3.6.1 An Automaton Equivalent to a Counterexample Trace

A phase event automaton will be used to represent a counterexample trace. While generating the automaton from the trace, the negation at the start of the counterexample formula is ignored. The formula which the generated phase event automaton will accept will in reality be the prohibited formula. Therefore, the satisfaction of the phase event automaton is the violation of the original formula.

3.6.2 A Trace Structure

To represent a counterexample formula, we will use a structure which represents a *trace*. Such a trace is a sequence of phase specifications.

$$\mathit{trace} \overset{\text{def}}{=} \text{seq } \mathit{PhaseSpec}$$

Each phase specification *PhaseSpec* represents a phase as defined in the counterexample formula syntax (see Section 3.3) together with its *entry events*. The entry events are the events which should occur before a phase can be entered. This is possible because an event is satisfied by an interval of length zero, thus all entry events should occur at the first instance of the interval being considered.

For example in the formula $\lceil A \rceil \frown \updownarrow B \frown \mathit{true}$, we have two phases separated by an event. The event is considered as the entry event of the second phase — it should occur at the first instance of this phase.

To mathematically define a phase specification we need a definition of time. This is defined as the set of positive reals:

$$\mathit{Time} \overset{\text{def}}{=} \mathbb{R}^{+}$$

Next, we define the time operations allowed for a phase:

$$\mathit{TimeOp} ::= \mathit{none} \mid \mathit{less} \mid \mathit{lessequal} \mid \mathit{greater} \mid \mathit{greatereaqual}$$

The structure of a phase specification is defined as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PhaseSpec}}\\
\mathit{inv} : \mathcal{L}(V)\\
\mathit{allowEmpty} : \mathbb{B}\\
\mathit{timeop} : \mathit{TimeOp}\\
\mathit{bound} : \mathit{Time}\\
\mathit{forbidden} : \mathbb{F}A\\
\mathit{entryEvents} : \mathcal{L}(A)\\
\hline
\mathit{allowEmpty} \Rightarrow (\mathit{inv} = [\mathit{true}] \wedge \mathit{timeop} \notin \{\mathit{greater}, \mathit{greaterequal}\})\\
\mathit{timeop} = \mathit{none} \Leftrightarrow \mathit{bound} = 0\\
\hline
\end{array}$$

inv is the predicate which must hold true *almost everywhere* in the phase. If no predicate is specified, then *inv* is $[\mathit{true}]$. For convenience we define *allowEmpty*, which is true if an empty interval satisfies the phase. Note that $\lceil \mathit{true} \rceil$ is not satisfied by the empty interval. *timeop* is the operator $\sim$ applied on the duration of the phase while *bound* is the constant. If there is no constraint on the length of the interval, then $\mathit{timeop} = \mathit{none}$ and $\mathit{bound} = 0$. *forbidden* is the set of forbidden events represented by the conjunction of $\boxminus ev$ while *entryEvents* is the conjunction of all *events* which should occur before the start of the phase.

3.6.3 Prefix

At this stage, it is useful to define the concept of a *prefix* of a formula. The automaton equivalent of a counterexample trace does not attempt to observe the trace as a whole. On the contrary this is done on a phase-by-phase basis. The function $\mathit{prefix}(i, t)$ returns a duration calculus formula which includes the phases up to and including the i^{th} phase of trace t. For example, let $t = \lceil A \rceil \frown \updownarrow B \frown \mathit{true}$. $\mathit{prefix}(1, t)$ returns the prefix $\lceil A \rceil$ while $\mathit{prefix}(2, t)$ returns the same formula as input since the formula has two phases.

Mathematically, *prefix* is defined as follows:

$$\begin{array}{|l}
\mathit{prefix} : \mathbb{N} \times \mathit{Trace} \rightarrow \mathit{DCForm}\\
\hline
\mathit{prefix}(0, t) = l = 0\\
\forall\, i : \mathbb{N} \mid i > 0 \cdot \mathit{prefix}(i, t) = \mathit{prefix}(i - 1, t)\\
\ \frown\ t(i).\mathit{entryEvents}\\
\ \frown\ (t(i).\mathit{allowEmpty} \vee \lceil t(i).\mathit{inv} \rceil) \wedge\\
\ell\ t(i).\mathit{timeop}\ t(i).\mathit{bound} \wedge\\
\bigwedge\{ev : t(i).\mathit{forbiddenEvents} \cdot \boxminus ev\}
\end{array}$$

To construct each phase from its phase specification, first there needs to be the zero-length interval which satisfies the entry events. This is followed by the interval which satisfies the invariant almost everywhere and the time constraint, or else it is satisfied by the empty interval. During this interval, it is ensured that no forbidden events occur.

3.6.4 Lowerbounds and Upperbounds

It is important to distinguish among lowerbound and upperbound phases because of the different role that the clock has in each case. A lowerbound phase is not *active* unless the clock has reached the specified constraint. On the other hand, an upperbound phase is only active while the clock has not reached its bound. For example, the lowerbound phase $\ell > 5$ has to wait for 5 time units for it to be observed. Conversely, for the upperbound phase $\ell < 5$, the phase is only active during the first 5 time units. The notion of an active and inactive phase is important and will be used over and over again in the following sections.

A phase is a lowerbound (LB) phase ($phase \in LB$) if and only if it has a lowerbound time constraint of the form $\ell > t$ or $\ell \geq t$. Similarly, a phase is an upperbound (UB) phase ($phase \in UB$) if and only if it has an upperbound time constraint of the form $\ell < t$ or $\ell \leq t$. This is defined as follows:

$$\begin{array}{|l}
LB, UB : Trace \rightarrow \mathbb{P}\,\mathbb{N} \\
\hline
\forall\, t : Trace \cdot LB(t) = \{i : \operatorname{dom} t \mid t(i).timeop \in \{greater, greaterequal\}\} \\
\wedge\ UB(t) = \{i : \operatorname{dom} t \mid t(i).timeop \in \{less, lessequal\}\}
\end{array}$$

3.6.5 Prefix Detection Situations

Recall that during the execution of an automaton a phase can be either active or inactive. In fact, this is not the complete story: it can be in any of the following four sets: *in*, *wait*, *gteq*, and *less*.

- A phase i is in the set *in* if it is currently *active*: the phases (prefix) up to the phase i have been observed and the phase i is still being observed.
- A phase is in *wait* if it is an element of *LB* and it is active, and the duration of the phase is less than its lowerbound. Hence, it is *waiting* for the clock to reach the bound before it can become active.
- A phase is in *gteq* if the time constraint is of the form $\ell \geq t$ and the phase is in *wait* ($gteq \subseteq wait$). In actual fact, there are cases where the time constraint is of the form $\ell \geq t$ but the phase is still not considered as a member of the set *gteq*. This issue will be explained in Example 3.6.1.
- A phase is in *less* if the bound is of the form $\ell < t$ and the phase is active and the duration is less than the bound. Yet, there are cases where the time constraint is of the form $\ell \leq t$ but the phase is still considered as part of the set *less*. This will be explained in Example 3.6.2. There is also another restriction on the phases in set *less*: $\forall\, i : less \cdot i - 1 \notin in \backslash wait\ \vee$

$\emptyset \nvDash tr(i).entryEvents$. This will be explained when seeping and clocks are discussed.

Relationship Between Sets

The relationship between the four sets can be summarised in the following schema:

$$
\begin{array}{|l}
\hline
\textit{PowerSet}(t : \textit{Trace}) \\
\textit{in}, \textit{wait}, \textit{gteq}, \textit{less} : \operatorname{dom} t \\
\hline
\textit{in} \subseteq \operatorname{dom} t \\
\textit{wait} \subseteq \textit{in} \cap LB(t) \\
\textit{gteq} \subseteq \textit{wait} \\
\textit{less} \subseteq \textit{in} \cap UB(t) \\
\forall\, i : \textit{less} \cdot i - 1 \notin \textit{in} \backslash \textit{wait} \vee \emptyset \nvDash t(i).\textit{entryEvents} \\
\hline
\end{array}
$$

The four sets can be shown as a single set by flagging the elements as follows:

- If phase $i \in in$ and $i \notin gteq, wait, less$ then the phase is represented by i in the single set.
- If phase $i \in wait \subseteq in$ and $i \notin gteq$ then $i \notin less$ and the phase is represented by $i^{>}$ in the single set.
- If phase $i \in gteq \subseteq wait \subseteq in$, then $i \notin less$ and the phase is represented by $i^{\geq}$ in the single set.
- If phase $i \in less \subseteq in$, then $i \notin wait, gteq$ and the phase is represented by $i^{<}$ in the single set.

If the phase does not belong to any of these four sets it is omitted from the single set.

3.6.6 Generating Locations

The locations of the generated phase event automaton are the power set of all the possible single sets according to the formula being translated. Each location is labelled by the single set of phases which it represents. For example, for the formula $\lceil A \rceil \frown \lceil B \rceil$ the following locations are generated: $\{\}$, $\{1\}$, $\{2\}$, $\{1, 2\}$.

Each location of the automaton has two invariants: one on the boolean variables and the other on the clock. These will be explained in more detail in the section about invariants: Section 3.6.15. However, at this point we need some explanation in order to allow the reader to understand other concepts. The clock invariant of any location is always in the form of a conjunction of $clock \leq bound$. The reason for not allowing $clock < bound$ is that we need a well defined time at which a location is left. In the case of $\mathit{clock} \leq \mathit{bound}$, the location is left when clock reaches bound.

Another issue is that once the automaton enters a location, it should stay there for some non-zero time interval. For this reason we only allow a *convex clock constraint*, i.e. a conjunction of upperbound clock constraints. Furthermore, a location is entered if and only if the clock valuations satisfy the *strict* version of the clock invariant of the destination location. The strict version is the clock constraint with the $\leq$ operators replaced by $<$.

3.6.7 Satisfying a Prefix

To define what it means for a prefix to be observed, the function *complete* is defined. This function takes a trace t, a location l, and a natural number i as inputs and returns the condition under which $\mathit{prefix}(i, t)$ is observed. There are basically three things which are relevant to observing a prefix ending with phase i:

1. Phase i is in the active set.
2. There are any clock bounds which have to be respected.
3. Phase i is satisfied by the empty interval, the entry events have been satisfied, and the prefix up to *i-1* has been satisfied.

One should agree that if the third condition is satisfied, then the prefix including phase i is being observed, even though it is not active. For this to hold we require that $i > 1$. Mathematically the condition can be written as:[4]

$$[i > 1 \wedge t(i).\mathit{allowEmpty}] \text{ `}\wedge\text{' } \mathit{complete}(t, l, i-1) \text{ `}\wedge\text{' } \mathit{tr}(i).\mathit{entryEvents}$$

If the conjunction of the first and second conditions holds, then the $\mathit{prefix}(i, t)$ is also being observed. In other words, a phase which is being observed is in the active set (first condition), but should not be in the *wait* set (waiting for the bound to expire). However, there are two exceptions (recall that all clock invariants are of the form: $\mathit{clock} \leq \mathit{bound}$): if $i \in \mathit{gteq}$ (even though by definition $i \in \mathit{wait}$) the phase is observed if $\mathit{clock} \geq \mathit{bound}$. Note that the last instance of an interval which satisfies an invariant $\mathit{clock} \leq \mathit{bound}$, also satisfied the constraint $\mathit{clock} \geq \mathit{bound}$. The other exception is that $\mathit{clock} < \mathit{bound}$ is not satisfied at the last instance of an interval which satisfies the clock constraint $\mathit{clock} \leq \mathit{bound}$. Therefore, $\mathit{clock} < \mathit{bound}$ has to be checked if the phase $i \in \mathit{less}$.

In summary, the condition to be checked if a phase $i \in \mathit{wait}$:

[4]Note that we use quotes to denote operators which are part of the final expression. Expressions within the square brackets are evaluated and their outcome will be part of the final expression.

```
    $if\ \ i \in l.wait\ \ then$
        $(if\ \ i \in l.gteq\ \ then\ \ c_i \geq t(i).bound$
        $else\ [false])$
    $else\ (if\ \ i \in l.less\ \ then\ \ c_i < tr(i).bound$
        $else\ [true]$
```

Putting all the parts together, the *complete* function is defined as follows:

```
$complete : Trace \times \mathrm{ran}\, PowerSet \times \mathrm{dom}\, Trace \rightarrow \mathcal{L}(A \cup C)$
-----
$\forall\, t : Trace;\ l : PowerSet(t);\ i : \mathrm{dom}\, t \cdot$
 $([i \in l.in]$ '$\wedge$'
     $if\ \ i \in l.wait\ \ then$
         $(if\ \ i \in l.gteq\ \ then\ \ c_i \geq t(i).bound$
         $else\ [false])$
     $else\ (if\ \ i \in l.less\ \ then\ \ c_i < t(i).bound$
         $else\ [true]))$
 $\vee\ ([i > 1 \wedge t(i).allowEmpty]$ '$\wedge$' $complete(t, l, i-1)$ '$\wedge$' $t(i).entryEvents)$
```

3.6.8 Moving Through Locations

When moving from a location to another we have to ensure that the active phases of the source location can *progress* into the phases of the destination location. This progression will be explained in the following sections through the notions of *seeping*, *keeping*, and *entering.*

3.6.9 Seeping

The concept of seeping is very important in observing a number of phases on chopped intervals. An automaton is said to *seep* through a phase if, while starting to satisfy a phase i, it also starts satisfying phase i+1. This is only possible if there are no events required to enter phase i+1. For example if $A \wedge B$ is true, the formula $\lceil A \rceil \frown \lceil B \rceil$ is said to seep through the first phase ($\lceil A \rceil$) and immediately also start satisfying $\lceil B \rceil$. For this reason, the function *canseep* is specified, which taking a trace, location and phase i, returns true only if i can be seeped into by phase i-1. Phase i can be seeped into, if and only if, phase i-1 is active and not waiting (hence being observed), and phase i has no entry events. This is defined as follows:

```
$canseep : Trace \times \mathrm{ran}\, PowerSet \times \mathrm{dom}\, Trace \rightarrow \mathbb{B}$
-----
$canseep(t, l, 1) = [false]\ \forall\, t : Trace;\ l : PowerSet(t);\ i : \mathrm{dom}\, t \mid i > 1 \cdot$
    $canseep(t, l, i) = [i - 1 \in l.in \setminus l.wait\ \wedge$
        $\emptyset \vDash t(i).entryEvents]$
```

Before seeping into a phase, it must be ensured that the predicate of the destination phase is satisfied. Therefore, to seep, it is not enough to check that the function *canseep* returns true, but there is also the condition that the predicate of the phase being seeped into, is satisfied. For this purpose, the function *seep* returns the condition to be satisfied in order to seep into a phase, given a trace, a location and a phase. Note that the condition has to be satisfied by the variables of the destination location (not the source location). For this reason we will mark variables with $'$.

$$\begin{array}{|l}
\mathit{seep} : \mathit{Trace} \times \operatorname{ran} \mathit{PowerSet} \times \operatorname{dom} \mathit{Trace} \rightarrow \mathcal{L}(V' \cup A \cup C) \\
\hline
\forall\, t : \mathit{Trace};\; l : \mathit{PowerSet}(t), i : \operatorname{dom} t \cdot \\
\quad \mathit{seep}(t, l, i) = [\mathit{canseep}(t, l, i)] \text{ `}\wedge\text{' } (t(i).\mathit{inv})'
\end{array}$$

Seeping and Clocks

Consider the formula:

$$\lceil A \rceil \frown \lceil B \rceil \wedge \ell < 5$$

So far we have observed that if both A and B are initially true, the second phase can be seeped into. However, one should note that the clock of the second phase should not be started unless B remains true and A turns to false. The reason is that, if $A \wedge B$ holds for more than 5 time units, the "extra" time can be "assigned" to the first phase. This explains why a clock should not be reset if a phase is entered through seeping (and not specifically entered). Another important conclusion is that a phase i should not be in the set *less* unless $\neg\, \mathit{canseep}(t, l, i)$ holds (where t is a trace and l is the location). This explains the condition $\forall\, i : \mathit{less} \cdot i - 1 \notin \mathit{in} \backslash \mathit{wait} \vee \emptyset \nvDash \mathit{tr}(i).\mathit{entryEvents}$ mentioned earlier which is equivalent to $\neg\, \mathit{canseep}(t, l, i)$.

3.6.10 Keeping a Phase Active

When going from one location (source) to another, a phase which was active in one location, can remain active in the destination location as well. A phase remains active if it is active in the source location, no forbidden events occur, and the invariant of the destination location is satisfied. Apart from these conditions we should check that if the phase is an upperbound, this has to be respected unless *canseep* is true (recall that when seeping the clock is ignored). If the phase is a lowerbound, we do not need to check anything since, if the phase is in *wait*, then it is also active and, if it is not in *wait*, then it remains active as well. Mathematically:

$$\begin{array}{|l}
keep : Trace \times \operatorname{ran} PowerSet \times \operatorname{dom} Trace \rightarrow \mathcal{L}(V' \cup A \cup C) \\
\hline
\forall\, t : Trace;\ l : PowerSet(t), i : \operatorname{dom} t \cdot \\
\quad keep(t, l, i) = [i \in p.in]\ `\wedge\text{'} \\
\quad\quad `\bigwedge\text{'}\ \{ev : t(i).forbiddenEvents \cdot \boxminus ev\}\ `\wedge\text{'}\ (t(i).inv)'\ `\wedge\text{'} \\
\quad\quad (if\ \ i \in UB(t) \wedge \neg\ canseep(t, l, i)\ \ then\ \ c_i < t(i).bound \\
\quad\quad else\ [true])
\end{array}$$

3.6.11 Entering a Phase

A phase can become activate when a transition is taken, in other words the phase is *entered*. To enter a phase i it must be ensured that the $prefix(i-1, t)$ has been completed, the entry events of phase i are satisfied, and the predicate of the destination location is satisfied. The function is given by:

$$\begin{array}{|l}
enter : Trace \times \operatorname{ran} PowerSet \times \operatorname{dom} Trace \rightarrow \mathcal{L}(V' \cup A \cup C) \\
\hline
enter(t, l, 1) = [false] \\
\forall\, t : Trace;\ l : PowerSet(t), i : \operatorname{dom} t \mid i > 1 \cdot \\
\quad enter(t, l, i) = complete(t, l, i - 1)\ `\wedge\text{'}\ t(i).entryEvents\ `\wedge\text{'}\ (t(i).inv)'
\end{array}$$

3.6.12 Special Cases of *gteq* and *less* Flags

There are certain cases where a constraint of the form $clock \leq bound$ has to be treated as $clock < bound$ and others of the form $clock > bound$ to be treated as $clock \geq bound$. To illustrate these situations the following examples are presented.

Example 3.6.1. Consider the formula:

$$\lceil A \rceil \frown \lceil B \rceil \wedge \ell \geq 5$$

If initially $A \wedge B$ holds, then by seeping, both phases start being observed. However, when 5 time units pass, these must belong to the second phase, while leaving the first phase with zero time. This is not acceptable and hence, when monitoring such a property, the location entered when initially $A \wedge B$ holds, should be labelled by $\{1, 2^{>}\}$ not $\{1, 2^{\geq}\}$.

Example 3.6.2. Consider the formula:

$$\lceil A \rceil \wedge \ell < 1 \frown \ell \leq 2$$

By considering the formula as a whole, one can immediately conclude that an interval whose length is greater or equal to 3 will not satisfy the formula since for the first phase $\ell < 1$. The problem when observing this formula is that the clock for the second phase is reset upon leaving the first phase, i.e. when $\ell = 1$.

In actual fact, the clock should have been reset when $\ell < 1$, but this is not a well defined time. The solution to this problem is to mark the second phase as *less* i.e. $\ell < 2$.

3.6.13 Creating Transitions

The purpose of transitions is to connect locations whose phases are related through *seeping*, *entering* or *keeping*. Transitions are also responsible for resetting clocks, and therefore, it has to be ensured that clock resets are done on the appropriate transitions. Each location has invariants, and hence, a transition should also ensure that the invariant of the destination location is satisfied before taking that transition.

When generating a transition, all the phases must be considered so that the generated automaton does not have any inconsistencies. In each of the following paragraphs we will explain the generation of a transition from location *l* to *l'*, on a trace *t*, with a set of clocks *X* to be reset :

All phases active in the destination location. A phase which is active in the destination location is either entered or kept from the source location, or seeped into in the destination location itself. If none of these are possible, then the phase in question cannot be active in the destination location. Mathematically:

$$\text{`}\bigwedge\text{'} \quad \{i : \operatorname{dom} t \cdot [i \in l'.in] \text{ `}\Leftrightarrow\text{'} \\ \quad keep(t, l, i) \text{ `}\vee\text{' } enter(t, l, i) \text{ `}\vee\text{' } seep(t, l', i)\}$$

Phases active in the destination location with lowerbound. In the case of lowerbound phase which is active in the destination location, a clock is reset if, and only if, $\neg\, keep(t, l, i)$ holds (where t is a trace, l a location, and i a phase). When *keeping* a phase the clock should not be reset; it should only be reset when the corresponding phase is *entered*. Furthermore, it cannot be an element of set *less*. Formally:

$$\text{`}\bigwedge\text{'} \quad \{i : \operatorname{dom} t \cdot \mathit{if}\ \ i \in LB(t) \cap l'.in\ \mathit{then} \\ \quad ([c_i \in X] \text{ `}\Leftrightarrow\text{' `}\neg\text{' } keep(t, l, i)) \text{ `}\wedge\text{' } [i \notin l'.less]\}$$

Phases marked as *wait*. A lowerbound phase, active in the destination location, can only be in *wait* if the clock is reset on the transition (i.e. no time has passed), or if the bound $c_i < bound$ still holds. This can be written as:

$$\text{`}\bigwedge\text{'} \quad \{i : \operatorname{dom} t \cdot \mathit{if}\ \ i \in LB(t) \cap l'.in\ \mathit{then} \\ \quad ([i \in l'.wait] \text{ `}\Leftrightarrow\text{'} \\ \quad\quad [c_i \in X] \text{ `}\vee\text{' } ([i \in l.wait] \text{ `}\wedge\text{' } c_i < t(i).bound)\}$$

Note that in the case of a clock reset we rely on the clock invariant (of the form $c_i \leq \mathit{bound}$) of the location as a condition to satisfy the clock constraint of the phase.

Phases marked as *gteq*. The phase is an element of *gteq* in the destination location if its time operation is $\geq$ and the reset did not occur too early due to seeping (see Example 3.6.1). There are no cases where $>$ is marked as $\geq$, only cases where $\geq$ is marked as $>$. Therefore, we restrict the phases marked as *gteq* as follows:

- Either the clock is reset, the phase has been entered, and the time operation is $\geq$. The reason is that the clock is not reset upon *keeping* and in the case of seeping the phase is not marked as $\geq$.
- Or if the clock was not reset, then the phase should have been an element of *gteq* in the source location and also an element of *wait* in the destination location (*gteq* $\subseteq$ *wait*).

In formal notation the conditions explained can be written as follows:

$$\begin{aligned}
&`\textstyle\bigwedge\text{'} \quad \{i : \mathrm{dom}\, t \cdot \mathit{if}\ \ i \in LB(t) \cap l'.\mathit{in}\ \ \mathit{then}\\
&\qquad ([i \in l'.\mathit{gteq}]\ `\Leftrightarrow\text{'}\\
&\qquad\quad \mathit{if}\ \ c_i \in X\ \ \mathit{then}\ [t(i).\mathit{timeop} = \mathit{greaterequal}]\ `\wedge\text{'}\ \mathit{enter}(t, l, i)\\
&\qquad\quad \mathit{else}\ [i \in l.\mathit{gteq} \wedge i \in l'.\mathit{wait}])\}
\end{aligned}$$

Phases active in the destination location with an upperbound. Recall that in the case of an upperbound phase which can be seeped into, there is no need to measure time. Hence, given an upperbound phase which cannot be seeped into in the destination location, the corresponding clock is reset if, and only if, the phase is entered or if had been seeped into in the source location.

We also have to ensure that any upperbound phase is not an element of *wait* or *gteq* in the destination location. In mathematics:

$$\begin{aligned}
&`\textstyle\bigwedge\text{'} \quad \{i : \mathrm{dom}\, t \cdot \mathit{if}\ \ i \in UB(t) \cap l'.\mathit{in} \wedge \neg\ \mathit{canseep}(t, l', i)\ \mathit{then}\\
&\qquad ([c_i \in X]\ `\Leftrightarrow\text{'}\ \mathit{enter}(t, l, i)\ `\vee\text{'}\ [\mathit{canseep}(t, l, i)])\\
&\qquad `\wedge\text{'}\ [i \notin l'.\mathit{wait}]\ `\wedge\text{'}\ [i \notin l'.\mathit{gteq}]\}
\end{aligned}$$

Phases marked as *less*. An upperbound phase is an element of *less* in the destination location if, and only if, it is not reset and was already marked as such in the source location, or if reset, it means that the phase either has a $<$ time constraint, or else, the phase has been marked as such because a reset came too late as in Example 3.6.2. To check for a late reset, it suffices to check that the phase could not have been *entered* and hence it was *seeped into* (it could not have been kept because it was already ensured that a reset only occurs upon an *enter* or a *seep*). Formally:

$$\text{`}\bigwedge\text{'}\quad \{i : \operatorname{dom} t \cdot \mathit{if}\ \ i \in \mathit{UB}(t) \cap l'.\mathit{in} \wedge \neg\ \mathit{canseep}(t, l', i)\ \ \mathit{then}$$
$$([i \in l'.\mathit{less}]\ \text{`}\Leftrightarrow\text{'}$$
$$\mathit{if}\ \ c_i \in X\ \ \mathit{then}\ [t(i).\mathit{timeop} = \mathit{less}]\ \text{`}\vee\text{'}\ \ \text{`}\neg\text{'}\ \mathit{enter}(t, l, i)$$
$$\mathit{else}\ [i \in l.\mathit{less}])\}$$

Phases with no clock constraints. In the case of a phase i which is neither a lowerbound nor an upperbound or for which *canseep* is true, it must be ensure that the corresponding clock c_i is not reset, and in the destination location, i is not an element of any of the sets: *wait*, *gteq*, and *less*.

$$\text{`}\bigwedge\text{'}\quad \{i : \operatorname{dom} t \cdot \mathit{if}\ \ i \notin \mathit{LB}(t) \wedge (i \notin \mathit{UB}(t) \vee \mathit{canseep}(t, l', i))$$
$$\wedge\ l'.\mathit{in}\ \ \mathit{then}$$
$$[c_i \notin X]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{wait}]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{gteq}]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{less}]\}$$

Putting all the above pieces together the function *guard* returns the condition of a transition, given a trace, initial location, set of clocks to be reset, and destination location. This is defined as follows:

```
$\mathit{guard} : \mathit{Trace} \times \operatorname{ran} \mathit{PowerSet} \times \mathbb{P}\, C \times \operatorname{ran} \mathit{PowerSet} \rightarrow \mathcal{L}(V' \cup A \cup C)$
------------------------------------------------------------
$\forall\, t : \mathit{Trace};\ l : \mathit{PowerSet}(t);\ X : \mathbb{P}\, C;\ l' : \mathit{PowerSet}(t) \cdot$
$\mathit{guard}(t, l, X, l') =\ \text{`}\bigwedge\text{'}\ \{i : \operatorname{dom} t \cdot$
    $([i \in l'.\mathit{in}]\ \text{`}\Leftrightarrow\text{'}$
        $\mathit{keep}(t, l, i)\ \text{`}\vee\text{'}\ \mathit{enter}(t, l, i)\ \text{`}\vee\text{'}\ \mathit{seep}(t, l', i))\ \text{`}\wedge\text{'}$
    $(\mathit{if}\ \ i \in \mathit{LB}(t) \cap l'.\mathit{in}\ \ \mathit{then}$
        $([c_i \in X]\ \text{`}\Leftrightarrow\text{'}\ \ \text{`}\neg\text{'}\ \mathit{keep}(t, l, i))\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{less}]\ \text{`}\wedge\text{'}$
        $([i \in l'.\mathit{wait}]\ \text{`}\Leftrightarrow\text{'}$
            $[c_i \in X]\ \text{`}\vee\text{'}\ ([i \in l.\mathit{wait}]\ \text{`}\wedge\text{'}\ c_i < t(i).\mathit{bound}))\ \text{`}\wedge\text{'}$
        $([i \in l'.\mathit{gteq}]\ \text{`}\Leftrightarrow\text{'}$
            $\mathit{if}\ \ c_i \in X\ \ \mathit{then}\ [t(i).\mathit{timeop} = \mathit{greaterequal}]\ \text{`}\wedge\text{'}\ \mathit{enter}(t, l, i)$
            $\mathit{else}\ [i \in l.\mathit{gteq} \wedge i \in l'.\mathit{wait}])$
    $\mathit{elseif}\ \ i \in \mathit{UB}(t) \cap l'.\mathit{in} \wedge \neg\ \mathit{canseep}(t, l', i)\ \ \mathit{then}$
        $([c_i \in X]\ \text{`}\Leftrightarrow\text{'}\ \mathit{enter}(t, l, i)\ \text{`}\vee\text{'}\ [\mathit{canseep}(t, l, i)])$
        $\text{`}\wedge\text{'}\ [i \notin l'.\mathit{wait}]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{gteq}]\ \text{`}\wedge\text{'}$
        $([i \in l'.\mathit{less}]\ \text{`}\Leftrightarrow\text{'}$
            $\mathit{if}\ \ c_i \in X\ \ \mathit{then}\ [t(i).\mathit{timeop} = \mathit{less}]\ \text{`}\vee\text{'}\ \ \text{`}\neg\text{'}\ \mathit{enter}(t, l, i)$
            $\mathit{else}\ [i \in l.\mathit{less}])$
    $\mathit{else}\ [c_i \notin X]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{wait}]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{gteq}]\ \text{`}\wedge\text{'}\ [i \notin l'.\mathit{less}])\}$
```

3.6.14 Initial Transitions

Creating the initial transitions requires the identification of locations from which the observation of the duration formula can start. Such a location must satisfy the following conditions:

1. All the phases with an upperbound should be waiting (i.e. no time has passed since the beginning).
2. The only phase which can be in the set *less* is the first one if its time bound is of the form *clock* $<$ *bound* (the subsequent phases cannot be in *less* because one of the conditions below is that *canseep* holds on phases which are not the first phases).
3. The only phases which can be in the *gteq* set are those whose time bound is of the form *clock* $\geq$ *bound*.
4. All the previous phases are *true phases* (with no predicate and no lower-bound) i.e. can be entered immediately.[5]

If all these conditions hold, then a phase is active if, and only if, the invariant holds, and the phase is the first one, or it can be seeped into. Else, if any of the above conditions fail, it means that the location does not represent the starting point of the observation of a formula. In summary this can be written as a function *init* which given a trace and a location, returns the condition to enter that location as the first location. This is defined as follows:

$$
\begin{array}{|l}
init : Trace \times \operatorname{ran} PowerSet \rightarrow \mathcal{L}(V') \\
\hline
\forall\, t : Trace;\ l : PowerSet(t) \cdot init(t, l) = \\
\quad if\ l.wait = l.in \cap LB(t) \wedge \\
\quad\quad l.less = if\ t(1).timeop = less\ then\ l.in \cap \{1\}\ else\ \emptyset \wedge \\
\quad\quad l.gteq = \{i : l.wait \cdot t(i).timeop = greaterequal \wedge \\
\quad\quad\quad (\forall\, j : 1..(i-1) \cdot t(j).allowEmpty)\} \\
\quad then\ \text{`}\bigwedge\text{'}\ i : \operatorname{dom} t \cdot \\
\quad\quad ([i \in l.in]\ \text{`}\Leftrightarrow\text{'}\ (t(i).inv)'\ \text{`}\wedge\text{'}\ ([i = 1 \vee canseep(t, l, i)])) \\
\quad else\ [false]
\end{array}
$$

3.6.15 Invariants for the Locations

There are two invariants for each location: one which concerns boolean variables and another which concerns clocks.

[5]Eg. $\lceil A \rceil \frown \lceil B \rceil \wedge \ell \geq 5$, if $A \wedge B$ are initially true, upon reaching length 5, the interval satisfying $\lceil B \rceil$ is given exactly 5 time units while $\lceil A \rceil$ is effectively given zero time which is incorrect.

The invariant concerning boolean variables is there to ensure that the predicates of all the active phases are satisfied and that the predicates of the phases which are not active but can be seeped into, are not satisfied. The second condition may seem to be useless but it is there to distinguish between $A \wedge \neg B$ and $A \wedge B$ in the case of the formula $\lceil A \rceil \frown \lceil B \rceil$. Consider the locations $\{1\}$ and $\{1, 2\}$. With the first condition alone, the invariant for $\{1\}$ is A while that for $\{1, 2\}$ is $A \wedge B$. If initially $A \wedge B$ is true, both invariants are satisfied and can be seeped into and this leads to non-determinism. Including the second condition the invariant for $\{1\}$ becomes $A \wedge \neg B$ which solves the problem of non-determinism. Formally we define the function s which given a location, returns its invariant:

$$s : \mathit{Trace} \times \operatorname{ran} \mathit{PowerSet} \to \mathcal{L}(V)$$

$$\forall\, t : \mathit{Trace};\ l : \mathit{PowerSet}(t) \cdot s(l) = \text{`}\textstyle\bigwedge\text{'}\ \{i : l.in \cdot t(i).inv\}\ \text{`}\wedge\text{'}$$
$$\qquad \text{`}\textstyle\bigwedge\text{'}\ \{i : \operatorname{dom} t \setminus l.in \mid \mathit{canseep}(t, l, i) \cdot\ \text{`}\neg\text{'}\ t(i).inv\}$$

The invariant concerning clocks has to include a bound ($\mathit{clock} \leq \mathit{bound}$) for the lowerbound phases which are in the *wait* set, and a similar bound for upperbound phases which cannot be seeped into. (Recall that there is no need to measure time for a phase which can be seeped into and has an upperbound.) This is defined by the function I which given a location, returns its clock invariant:

$$I : \mathit{Trace} \times \operatorname{ran} \mathit{PowerSet} \to \mathcal{L}(V)$$

$$\forall\, t : \mathit{Trace};\ l : \mathit{PowerSet}(t) \cdot I(l) = \text{`}\textstyle\bigwedge\text{'}\ \{i : l.in \mid i \in l.wait\ \vee$$
$$\qquad (i \in \mathit{UB}(t) \wedge \neg\ \mathit{canseep}(t, l, i)) \cdot c_i \leq t(i).bound\}$$

3.6.16 The Complete Automaton

The complete automaton is given by the following schema:

$$\mathcal{P}(t : \mathit{Trace})$$
$$P \stackrel{\text{def}}{=} \mathit{PowerSet}(t)$$
$$C \stackrel{\text{def}}{=} \{i : \mathit{UB}(t) \cup \mathit{LB}(t) \cdot c_i\}$$
$$E \stackrel{\text{def}}{=} \{l : P;\ X : \mathbb{P}(C);\ l' : P \mid \mathit{guard}(t, l, X, l') \neq [\mathit{false}]$$
$$\qquad \cdot (l, \mathit{guard}(t, l, X, l'), X, l')\}$$
$$s : P \to \mathcal{L}(V)$$
$$I : P \to \mathcal{L}(C)$$
$$E_0 \stackrel{\text{def}}{=} \{p : P \mid \mathit{init}(t, l) \neq [\mathit{false}] \cdot (\mathit{init}(t, l), p)\}$$

3.6.17 Example

For an example we consider the formula $\neg\ (\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil)$.

Locations

First we start by generating the locations. The first phase is a lowerbound phase and hence it starts in the *wait* set. Using the *PowerSet* schema defined above, we get the following locations:

$$\{\emptyset, \{1\}, \{1^>\}, \{2\}, \{1,2\}, \{1^>, 2\}\}$$

Clocks

By the definitions of UB and LB respectively:

$$UB = \emptyset$$

$$LB = \{1\}$$

Hence, the only clock required for the example is the clock for the first phase c_1.

Invariants

Using the functions to generate the invariants and clock invariants we show the result for some of the locations:

For the location $\{1\}$ we get the invariant $A \wedge \neg B$ since the first phase is active while the second is not, but *canseep* is true. On the other hand, *canseep* does not hold for the second phase of $\{1^>\}$ since the phase is *waiting.*

$$s(\{1\}) = A \wedge \neg B$$

$$s(\{1^>\}) = A$$

The clock invariant only applies for the location $\{1^>\}$ where the first phase is *waiting.* Using the function I we get the clock invariant as follows:

$$I(\{1^>\}) = c_1 \leq 1$$

Initial Transitions

Applying the *init* function on the locations, only two locations have an initial transition. For example location $\{2\}$ cannot have an initial transition because the active phase 2 is neither the first phase, nor can it be seeped into. Thus, it cannot be the first location at which the automaton starts (the bi-implication cannot be satisfied). As a positive example consider the empty set. Intuitively the automaton can start in the empty set if initially A is false i.e. the formula can never be satisfied if initially A is false. Using the mathematical formula: since

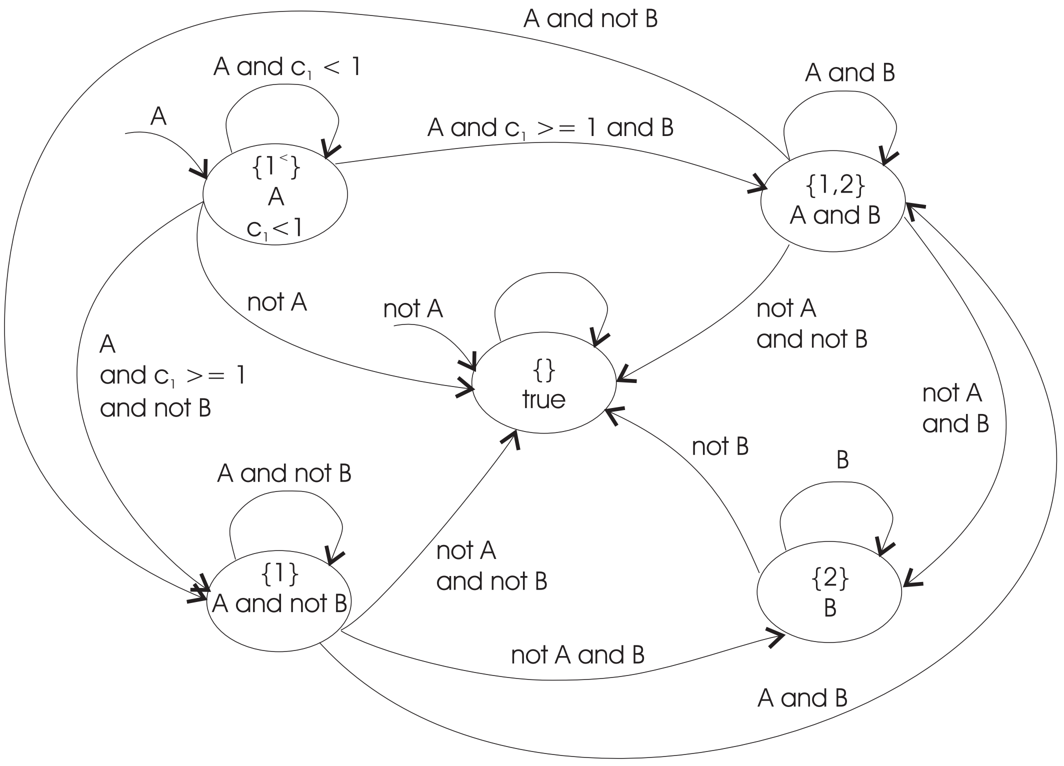


Figure 3.2: The phase event automaton equivalent to the formula $\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil$.

phase 1 is not active in location $\emptyset$ it must be ensured that A is false. For the second phase the bi-implication is already satisfied because phase 2 is not the first phase and cannot be seeped into.

Transitions

We now give some examples of transitions. For example consider the transition from $\{1\}$ to $\{1, 2\}$. It is the point where A has been true for more than one time unit and B is false. As soon as B turns true, both *prefix*(1) and *prefix*(2) will start to be satisfied. This means that phase 1 will be *kept* while phase 2 will be *entered* and *seeped* into. The condition for this transition is directly obtained from the functions *keep*, *enter*, and *seep* which are all equal to the invariant of the destination location: $A \wedge B$.

As another example we take the transition from $\emptyset$ to $\{1, 2\}$. Even though phase 2 can be *seeped* into, phase 1 cannot be *kept*, *entered*, or *seeped* into. Therefore, the transition can never be taken i.e. the guard is false.

Result

The equivalent phase event automaton is shown in Figure 3.2.

3.7 Conclusion

In this chapter, we have introduced duration calculus which is a highly expressive dense time logic. In fact, in general it is too expressive to monitor. Thus, various subsets of duration calculus have been studied in the literature. In the above sections, we have described two such subsets: QDDC and counterexample traces. By limiting ourselves to counterexample traces, monitoring is possible by translating counterexample traces into phase event automata and then using these automata as monitors. For this reason, we have explained in details the translation from counterexample traces to phase event automata. Similarly, the QDDC subset has been presented because it can be monitored by symbolic automata using the Lustre syntax.

Undoubtedly there are many other real-time logics besides those described above, for example interval temporal logic [71], timed linear temporal logic [5], metric temporal logic [64] and timed regular expressions [9]. We choose duration calculus mainly because it is an interval temporal logic. This means that it is based on intervals rather than points in time. For runtime verification it is a significant advantage that there is no distinction between past and future. For this reason and its sound mathematical background, duration calculus is ideal to represent real-time software properties. The advantage of duration calculus over other interval-based real-time logics is that it gives a structure to temporal variables using an integral operator on state variables [49].

Part II

Larva

4. Dynamic Automata with Timers and Events (DATE)

4.1 Introduction

In the previous chapters, we have introduced dynamic analysis with special emphasis on runtime verification, and we introduced various real-time logics. It is not always easy to represent practical security properties using the reviewed logics. From our experience we felt the need for a concise logic to express properties for real-time systems. The logic should not only have timers to represent real-time properties but should also be dynamic so that properties can be verified for each individual object. Another feature of the logic are channels used to allow properties to communicate with each other. Furthermore, the new logic is devised with the aim of providing the framework for more guarantees on the effect of monitoring. With this expressive logic and guarantees we aim to provide a practical system which assists developers in creating more reliable and robust software.

This chapter is organised as follows: first we establish the design aspects of our logic in Section 4.2 through a detailed discussion based on the background research provided in the previous chapters. Then, we give the mathematics of the logic in Section 4.3 and we conclude in Section 4.4.

4.2 Design Choices of a Runtime Verification Logic

Static Analysis vs Dynamic Analysis

For our intents and purposes, dynamic analysis is more attractive than static analysis methods such as model checking because it scales up well. Furthermore, it allows more freedom of expressivity since there is no need for the logic to be decidable over all execution paths. To understand this point one should understand that in model checking a property needs to be satisfied by all the possible execution paths. On the other hand, for dynamic analysis one only needs to check the satisfaction of the property for the current execution. Being able to check a property for a single execution path requires a different technique and much less computational resources. For this reason the limit on expressivity is much less for dynamic analysis than for model checking.

It would have been very interesting to integrate static analysis and dynamic analysis in one system. For example static analysis could have been very useful to intelligently generate test cases for a dynamic analysis tool to find errors during testing [8]. However, we have to focus our research because of many limitations and hence we will stick to dynamic analysis.

Choosing the Dynamic Analysis Flavour

The assertions in design by contract are made manually in various locations in the system code. This immediately introduces the possibility of errors since humans can easily omit or misplace some assertions. Ideally, the assertions are not manually inserted into the system, but rather automatically weaved. Moreover, we would like the security properties to be centralised rather than scattered throughout the code. Changing a security property from a central location is much easier and less error-prone. Considering these issues in design by contract, we will not adopt this approach in our research.

Runtime verification is much more appropriate because it uses formal notation for specifying security properties. Formal notation is more succinct and abstract than the actual implementation and this makes it less error-prone — it is easier to make mistakes in the low-level implementation details rather than in the high-level description of a property. Another attractive aspect of runtime verification is that it allows the user to specify extra code so that the system which finds itself in a bad state, can revert itself back to a valid state. This is very desirable because it eliminates the need of human intervention upon a security violation. Developers may not like a verification system to be intrusive, therefore it should be left up to the developer to decide what the recovery actions might be.

Exploring the possibility of having the violation mechanism as part of the system design, as designated by monitoring-oriented programming, is a very interesting area. However, in our research we plan to limit ourselves to treat the verification as a “double-check” rather than the actual check. This decision is mainly due to the fact that the systems under our consideration have already been designed and implemented. Furthermore, there is the issue of synchronous and asynchronous verification which has already been mentioned as online versus offline verification (Section 2.3.5). However, we can also explore the possibility of finding a compromise between total synchrony and completely offline verification. This can be achieved by allowing a possibility of a delay between the verification system (which can be run on a separate machine) and the target system. Furthermore, the delay allowed may be dependent on the criticality of the part of the system in which we are executing. Therefore, in a critical section we may wait for the verifier to synchronise, while in a non-critical section we can afford to allow the verifier to run asynchronously. These ideas will not be developed further in this work because they are outside the scope.

Having an explanation of how a bad state was reached, as suggested in runtime reflection, is very desirable. This is more especially so when dynamic analysis is used during the testing phase to identify errors. This idea is taken up and somehow we would like to provide a means for the user to understand what went wrong in the system.

Logics

Choosing the logic which is most appropriate for our project is not trivial. Linear temporal logic is arguably very easy to understand as a logic but it is not expressive enough for our intents and purposes, since we have the target of verifying real-time properties. To this end there is TLTL [5] which extends LTL with real-time. The problem remains that LTL refers to particular points in time and distinguishes between the past and the future. This creates some problems in runtime verification. Although there are ways [11, 33, 30] to go about this issue, we opt for logics which refer to intervals rather than points in time. For this reason we investigate duration calculus. However, duration calculus poses a number of difficulties to implement monitors for it. It is also not trivial to write properties with duration calculus. Therefore, we will try to investigate a subset of duration calculus which is implementable and for which we can use easier notation or syntactic sugaring.

Regular expressions may be much more desirable because they are so widely used in other applications and therefore users may already be familiar with them. Nonetheless, users may still find it strange to use regular expressions to represent system properties. Another issue is that given a regular expression, a user will not know whether that is an accepted sequence of events or a prohibited sequence. Apart from these usability issues there are also expressivity issues. We may not only be interested in a sequence of events but also in some parameters related to those events. For example, these parameters can be used to check a condition before taking a transition. Hence, we require something which can accept parametrized events.

This leads us to consider the use of automata to which we can add our own extensions for the necessary expressivity. Starting with very little expressivity and increasing it as the need arises from the practical case studies that we consider has the advantage of avoiding any unnecessary expressivity which will only result in more complexity without adding any benefit. Automata also have the advantage of being pictorial. This may be very useful for the user to visualise the requirements. There are various automata which we can use as our basis to which we can add the extensions. However, some of them were automatically eliminated because of their special features which are not applicable for our intents and purposes. A case in point are Büchi automata (as used by Courcoubetis et al. [26])

which are intended for infinite strings. When we perform runtime verification, we can never have an infinite string because by definition, runtime verification works on a running program which cannot produce an infinite string of events. Another example are alternating automata (as used by Finkbeiner and Sipma [37]) which have *and* and *or* transitions. These types of transitions are useful to represent logics which have the boolean *and* and *or* but this is not our case.

Apart from the need of parametrized events, we also need to have local variables to store the necessary values. Without such variables, a simple automaton has to become much larger. For example, consider the automata in Figure 4.1 and Figure 4.2.

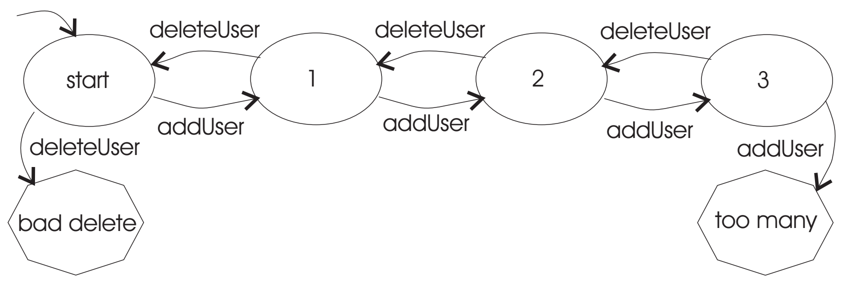


Figure 4.1: An automaton monitoring the adding and deleting of users without a variable.

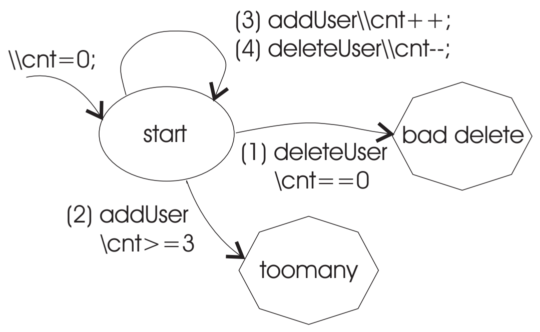


Figure 4.2: An automaton monitoring the adding and deleting of users with a variable.

The two automata actually both do the same thing, but one is much smaller than the other because it uses a variable. In fact the variable allows us to encode a very large (possibly infinite) automaton into the small automaton which is easier to visualise and understand. Such automata (which use variables) are known as symbolic automata. Therefore we will use symbolic automata as the basic notation to which we will add any necessary extensions.

Customisation of Symbolic Automata

A symbolic automaton has an alphabet, a set of states, a set of variables and a set of transitions. The following are the customisations which we need for our verification architecture.

Events The alphabet of the automaton are the events. The events we handle are method calls and exception throws. Each event can have any number of variables which are given a value upon the occurrence of that event. The assigned value is usually related to the context of the method call. For example, it can be directly bound to the parameters, target, or return object of the method. However, the variable can also be assigned any arbitrary value.

Conditions It is useful to be able to distinguish between occurrences of the same event. For example an addition of a user may trigger a transition to a good state if the the total number of users is acceptable, but to a bad state otherwise. Therefore, for each transition we not only need to specify the event, but also the condition. If the condition does not hold, the transition is not taken. If no condition is specified it is assumed that the transition should be taken in any case. The condition can have access to both the automaton's local variables and the parameters of the transition's event.

Actions Having the automaton's local variables is of little use if we cannot manipulate them. To this extent we provide the possibility of an action over a transition. This action may consist of one or more statements which can manipulate the automaton's variables. An action has also read-only access to the event's (on which the transition triggers) parameters and access to the underlying system's variables according to their modifiers (public, static, or private). Therefore, the action may also be used to perform operations using these variables. One should note that the action is not executed unless the condition of the transition is satisfied. At this point we would like to highlight the extent of the expressive power we give the user through the monitoring system: the monitoring system does not only fully control its own state but can also modify the state of the system being monitored. This is very powerful but also very risky. It is the user's responsibility to ensure that this power is correctly used without excessive (and possibly dangerous) intrusion on the monitored system.

Actions generating Events An interesting issue is whether we should allow actions to generate events. If we do so, there is the possibility of transitions triggering each other. These are known as *chained transitions*. Once transitions start triggering each other, there is no guarantee that no infinite loop is formed. To avoid this, in our architecture, actions cannot generate events.

Code in each State The possibility of putting an action on a transition is very useful, but what if the action is more related to the state, rather than the transition? In such a case all the incoming or outgoing transitions will have the same action (or at least a part of it will be the same). To solve this issue, we provide actions in states. This option will eliminate any duplication of actions in the incoming or outgoing transitions of a particular state. The action will take place exactly upon entry into that state. In other words, after the execution of the action of the incoming transition.

Initialisation An important issue which we must consider is where we can place the initialisation of the automaton's variables. Conceptually, it should occur before the automaton enters the first state and this idea was adopted in our automata. The initialisation can also be used to restore the state of the automaton from a previous run of the application. This idea is useful in practice but is not included in the system architecture because this is not considered as a central part of the system.

Bad States Somehow, we need to distinguish among states so that we are able to identify bad behaviour. Therefore, we require the notion of bad states such that if a trace reaches a bad state, this means that the property being verified has been violated.

Accepting States In order to identify automata which have completed their monitoring and can be considered as obsolete, we require states called *accepting*. Once an automaton reaches an accepting state, no other states can be taken and the automaton stops.

Contexts We sometimes need to have a separate automaton for each particular object. We also need some way of distinguishing which object generated a particular event so that only the corresponding automaton is triggered. However, it is still desirable to have common (global) variables which the automata can update and access. These global variables will in turn be accessible to a global automaton which does not belong to a particular object. We extend this notion further by allowing more than one object to be specified for each automaton. We will call these objects the *context* of the automaton. For example, a context can be made up of a *colour* and a *shape* object. The context of an automaton is what distinguishes it from other automata. No two automata can have the same context.

Apart from having a two-level hierarchy of global and non-global (contextual) variables, we can have different levels of context. We will refer to this as *nesting*. For example, we can have a global automaton which represents the *bank* context, a more specific automaton for each *user*, and a yet more specific automaton for each *account* of each *user*. Similar to the concept of global variables, the more specific context will always have access to the higher level variables. In the above example, from the context of the *account* we can have access to the variables in *user*, but from the *user* context we cannot access the variables in *account* — from the user level we do not have the context of one particular account (a user may have many accounts).

Events from other Automata through Channels Having global variables provides a limited way of communication among the automata. To provide a higher level of communication we can allow an automaton to trigger events to other automata. This concept of communication will be referred to as *channel* communication. A practical application would be the monitoring of a large program with many user functions. Some patterns of user behaviour are considered malicious and the user in question should be blocked from the system. Such patterns may be the sequence or frequency of particular function invocations. For this reason, specialised automata (for each behavioural pattern) increment a common global counter, while a global automaton monitors that counter. If the counter reaches a particular threshold, then an action is carried out by the general (global) automaton. To implement this, we can either use polling on the global counter, or else, we need communicating automata. The former approach is not efficient, but the latter is — each time the counter is incremented, the global automaton is notified by the specialised automaton which incremented the counter.

As with the problem of chained transitions, we need to be careful regarding the allowed sequence of communication among automata. This is because we can cause an infinite sequence of events and the communicating automata will end up looping forever.

Invariants Another concept which is very useful in many practical applications is the concept of invariants. Recall that we can create an automaton for each individual object that we are monitoring. Usually, this object should be only modified in previously known patterns. For example, we may know that along a sequence of states, a certain attribute of the monitored object should not change. Consider the identification field of a transaction which should remain fixed once it is set. In this case, the identification field is an invariant. If any of the invariant restrictions are violated, the automaton should revert to a bad state. The checking is done before taking a transition i.e. moving from one state to the next. This will successfully monitor whether the object has been illegally modified while the automaton was in a particular state.[1]

[1]Note that the state of the verifying automaton does not correspond to one state of the executing program being monitored. Therefore, many operations could have been done on the object during one state of the monitoring system.

Real-time In many practical applications, we need to measure the real-time between events. For example we have to ensure that customers get processed within a particular period since their application for a service. For this reason we need clocks as used in integration automata [1]. We propose allowing the user to use a set of clocks, where each clock can generate events, respond to queries and perform actions. A clock event should be able to trigger a transition when the clock reaches a particular time. Furthermore, the queries will be of the form *clock operator amount*, where the clock value is compared to a quantity. The basic actions which the clock should perform are *reset*, *pause* and *resume*. Having these constructs we believe that a lot of interesting properties can be specified.

4.3 The Logic

In this section, we present a theory of communicating automata with events and timers. Each mathematical construct introduced will be related to the requirements explained in the previous section.

4.3.1 Dynamic Automata with Timers and Events

Events

The underlying logic we will use to define properties will be based on communicating symbolic automata with timers, whose transitions are triggered by events. Events are built as a combination of visible system actions. The possible types of events are: system events (method calls or exception throws), timer events, and channel communication (automata synchronisation).

Definition 4.3.1. Given a set *systemevent* of events which are generated by the underlying system and may be captured by the runtime monitors, a set of timer variables *timer*, and a set of *channels*, a composite *event* made up of system events, channel synchronisation, a timeout on a timer, a choice between two composite events (written $e_1 + e_2$), or the complement of a composite event (written $\overline{e}$), is syntactically defined as follows:

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

We say that a basic event x (which can be a system event, a channel synchronisation or a timeout event) will fire a composite event expression e (written $x \vDash e$) if either (i) x matches exactly event e; or (ii) $e = e_1 + e_2$ and either $x \vDash e_1$ or $x \vDash e_2$; or (iii) x is a system event and $e = \overline{e_1}$, and $x \nvDash e_1$.

Example 4.3.1. A basic event *bad_login* and a composite event *logout*, where

$$\begin{aligned} logout &= inactive\_clock \ @ \ 5 + illegal\_action \\ illegal\_action &= bad\_login + restricted\_access \end{aligned}$$

Using the rules above, *bad_login* will cause *logout* to trigger — first it causes *illegal_action* to trigger and this, in turn, causes *logout* to trigger (applying above rule $e = e_1 + e_2$ twice).

The notion of firing of events can be extended to work on sets of events. Given a set of basic events X a composite event e will fire (written $X \vDash e$) if either (i) e is a basic event expression and for some event $x \in X$, $x \vDash e$; or (ii) $e = e_1 + e_2$ and either $X \vDash e_1$ or $X \vDash e_2$; or (iii) X contains at least one system event and $e = \overline{e_1}$, and for all $x \in X$, $x \nvDash e_1$.

The semantics of the complement of an event is constrained to fire when at least one system event fires, so as to avoid triggering whenever a timer event or channel communication happens, thus making such events to necessarily depend on the underlying system. This constraint can be relaxed without affecting the results in this work.

Example 4.3.2. A set of events X and a composite event *illegal_action*, where

$$\begin{aligned} X &= \{\mathit{an\_illegal\_action}\} \\ \mathit{illegal\_action} &= \mathit{bad\_login} + \mathit{restricted\_access} + \overline{\mathit{a\_legal\_action}} \end{aligned}$$

In this example, *illegal_action* will trigger in the following way: since the only event in set X is neither the basic event *bad_login* nor the basic event *restricted_access* these will not cause the triggering. However, since the event is not *a_legal_action*, the complement construct will cause the composite event *illegal_action* to trigger.

A special event *init* triggers at the start of the monitoring system. The purpose of this event is to allow more flexibility in the initialisation of automata. Later on, this event will be specifically useful for translating phase event automata into DATEs. In practical terms, the *init* event can be defined in terms of a clock which generates an event automatically at the start of the monitoring system.

Symbolic Timed Automata

In what follows, we will call *symbolic timed automata* a certain class of automata with timers. However, such timers allow reset, pause and resume actions as in integration automata [1]. Thus, these should not be confused with Alur and Dill's timed automata [4]. These timers enable us to meet the design decisions discussed in the previous section (under the header *real-time*), allowing us to measure the duration between events.

Definition 4.3.2. The *configuration of the system timers* (written $\mathcal{CT}$) is a function from timers to (i) the time value recorded on the timer; and (ii) the state of the timer (running or paused).

Timer resets, pauses and *resumes* are functions from a timer's configuration to another, changing only the value of one timer to zero (in the case of a reset), or the state of one timer (in the case of pause or resume). A *timer action* (written $\mathcal{TA}$) is the functional composition of a finite number of resets, pauses and resumes.

Properties of a given system will be expressed as communicating timed automata. These automata will have access to read and modify the state of the underlying system. (This is also according to our design choices regarding *actions* in the previous section.)

Definition 4.3.3. A *symbolic timed automaton* running over a system with state of type Θ is a quintuple $\langle Q,\ q_0,\ \rightarrow,\ B,\ A\rangle$ with set of states Q, initial state $q_0 \in Q$, transition relation $\rightarrow$, bad states $B \subseteq Q$, and accepting states $A \subseteq Q, A \cap B = \emptyset$. Transitions will be labelled by (i) an event expression which triggers them; (ii) a condition on the system state and timer configuration which will enable the transition to be taken; (iii) a timer action to perform when taking the transition; (iv) a set of channels upon which to signal an event; and (v) code which may change the state of the underlying system:

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

We will assume that a total ordering $<$ exists on the transitions to ensure determinism.

The behaviour of an automaton M upon receiving a set of events consists of (i) choosing the highest priority transition which fires with the set of events and whose condition is satisfied; (ii) performing the transition (possibly triggering a new set of events); and (iii) repeating until no further events are generated, upon which the automaton waits for a system or timeout event.

Definition 4.3.4. For a symbolic timed automaton M, we say that a set of system scheduled events X, system state $\theta \in \Theta$, timer configuration T and state q (in which M currently resides), *performs a step* to X', θ' and q', with timer update t' (written $(X, \theta, q) \Rightarrow^T_{t'} (X', \theta', q')$) if $q \notin A$ and $(q_1,\ e,\ c,\ t,\ O,\ f,\ q_2)$ be the largest (in terms of $<$) transition in $\rightarrow$ such that: (i) $q = q_1$; (ii) $X \vDash e$; (iii) $c(\theta, T)$, and the following hold: (i) $t' = t$; (ii) $q' = q_2$; (iii) $\theta' = f(\theta)$; (iv) $X' = O$. If no such transition exists, we write $(X, \theta, q) \Rightarrow^T_{id} (\emptyset, \theta, q)$.
The notion of automata performing a step can be extended over a vector of automata communicating via broadcast channels. Given a vector of n automata $\bar{M} = \langle M_1, M_2, \ldots M_n\rangle$, in states $\bar{q} = \langle q_1, q_2, \ldots q_n\rangle$ and with shared timers in state T, we write that $(X, \theta_0, \bar{q}) \Rightarrow^T_{t'} (X', \theta_n, \bar{q}')$ if (i) for each $1 \leq i \leq n$, $(X, \theta_{i-1}, q_i) \Rightarrow^{T'_i}_{t'_i} (X'_i, \theta_i, q'_i)$; (ii) $t' = t'_n \circ t'_{n-1} \circ \ldots \circ t'_1$; (iii) $X' = X'_1 \cup X'_2 \cup \ldots X'_n$; (iv) $T'_i = (t'_{i-1} \circ t'_{i-2} \circ \ldots \circ t'_1)(T)$; (v) $T' = t'_n(T'_n)$; and (vi) $\bar{q}' = \langle q'_1, q'_2, \ldots q'_n\rangle$.

Note that the order of execution is set by the order of the automata, once again to avoid non-determinism. Clearly, as in any programming with side-effects, the use of actions on the transitions must be carefully handled. Also note that the timer actions are accumulated so as to evaluate all conditions with the same initial timestamps.

Since no transition can be taken from an accepting state $q \in A$, once the automaton reaches such a state, it cannot change its state.

Example 4.3.3. Consider a system where one needs to monitor the number of successive bad logins and the activity of a logged-in user. By having access to *badlogin*, *goodlogin* and *interact* events, one can keep a successive bad-login counter and a clock to measure the time a user is inactive. Figure 4.3(a) shows the property that allows for no more than two successive bad logins and no more than 30 minutes of inactivity when logged in, expressed as a DATE. Upon the third bad login or 30 minutes of inactivity, the system reverts to a bad state. In the figure, transitions are labelled with events, conditions and actions, separated by a backslash. It is assumed that the bad login counter is initialised to zero.

Figure 4.3(b) shows how actions can be used to remedy the situation (when possible), instead of going to a bad state. For example, after too many bad logins, one can block the user from logging in for a period of time, and upon 30 minutes of inactivity, the user may be forced to logout.

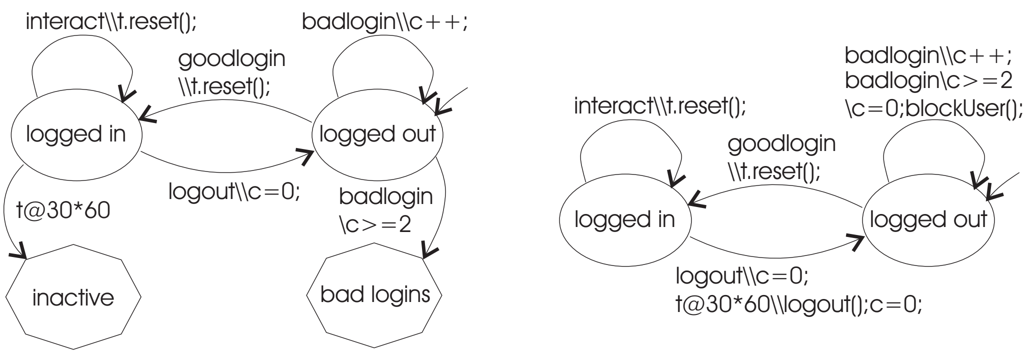


Figure 4.3: (a) An automaton monitoring the bad logins and activities occurring in a system; (b) The same automaton with recovery actions.

Example 4.3.4. Building on the previous example, we will show the use of channels for automata communication. In this example, a good login event will not be considered as a system event, but rather as a property being monitored by a separate automaton. The only change in the automaton monitoring successive bad logins (in Figure 4.3) is to listen for an event on channel *ChGoodlogin* rather than the system event *goodlogin*. The automaton which monitors good logins and sends an event on the channel *ChGoodlogin* is shown in Figure 4.4. One should notice that there are two outgoing transitions with the same event (*PressOK*) from state *Prompt for PW*. The transition with the higher priority is the one with the condition *checkPassword*. If this fails, then the other transition is taken.

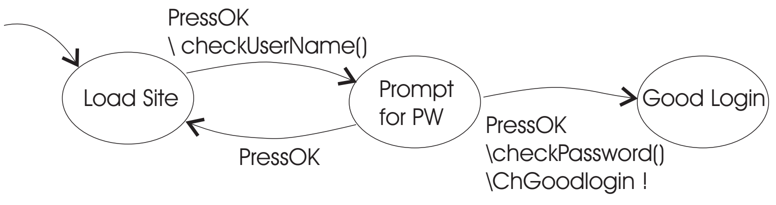


Figure 4.4: An automaton monitoring the good logins and sending events on channel *ChGoodlogin*.

Dynamic Automata with Timers and Events

The notions of symbolic timed automata can be lifted to work on dynamic networks of symbolic timed automata, in which we enable the creation of new automata during execution in a structured manner. This logic will be referred to as Dynamic Automata with Timers and Events (DATE).

Definition 4.3.5. A DATE $\mathcal{M}$ is a pair $(\bar{M}_0, \nu)$ consisting of (i) an initial set of automata $\bar{M}_0$; and (ii) a set of automaton constructors ν of the form:

$$\mathit{event} \times (\Theta \times \mathcal{CT} \to \mathbb{B}) \times (\Theta \times \mathcal{CT} \to \text{Automaton})$$

Each triple $(e, c, n) \in \nu$ triggers upon the detection of event e, with the state and timer configurations satisfying condition c, and creating an automaton using function n. The triggered automata in time configuration T, with events X, in system state θ (written $tr(T, X, \theta)$) is defined to be:

$$tr(T, X, \theta) \stackrel{\text{def}}{=} \{n(\theta, T) \mid (e, c, n) \in \nu,\ X \vDash e,\ c(\theta, T)\}.$$

Finally, the events created by the transition actions, can themselves trigger new transitions.

When constructing new automata, there is an important issue which we have not yet considered. Recall that there are two types of variables: global variables and local variables. Upon constructing an automaton, a set of new local variables must be created as well. The new variables should then be passed to the new automaton as parameters so that these variables replace the existing ones. Thus, the function which constructs automata will be of the form:

$$\Theta \times \mathcal{CT} \to (\Theta \to \text{Automaton})$$

The same idea should be used for non-global clocks and channels. Furthermore, a structure should be used so that when the constructed automaton reaches an accepting state and is removed, the new variables which were created with the construction, can also be identified and removed.

Definition 4.3.6. The configuration of a DATE contains: (i) the state of the timers; (ii) the state of the underlying system; and (iii) the state of the currently running automata — a vector of a state for each automaton in the network.
A DATE is said to *perform a full-step* from configuration $(T, \theta, \bar{q})$ to configuration $(T', \theta', \bar{q}')$, upon receiving a set of system actions X, (written $(T, \theta, \bar{q}) \not\Rightarrow_X (T', \theta', \bar{q}')$) if for some number n:

$$(X_0, \theta_0, \bar{q}_0) \Rightarrow^{T'_1}_{t_1} (X_1, \theta_1, \bar{q}_1) \Rightarrow^{T'_2}_{t_2} \ldots (X_n, \theta_n, \bar{q}_n) \Rightarrow^{T'_{n+1}}_{t_{n+1}} (\emptyset, \theta_{n+1}, \bar{q}_{n+1}),$$

where: (i) $X = X_0$, $\bar{q}_0 = \bar{q}$, $\theta = \theta_0$ and $\theta' = \theta_{n+1}$; (ii) the final state of the timer is updated according to the timer's accumulated actions ($1 \leq i \leq n+1$): $T'_i = (t_{i-1} \circ t_{i-2} \circ \ldots \circ t_1)(T)$; (iii) $T' = t_{n+1}(T'_{n+1})$; and (iv) the initial states are updated as required by DATE triggers $\bar{q}_i = \bar{q}_{i-1} \oplus \bigcup_j q_{0_j}$ where $\bigcup_j q_{0_j}$ is the set of initial states — one for each automaton in $tr(T'_i, X_i, \theta_i)$.
Such a step is called a *good full-step*, if no bad states appear in the intermediate state vectors.

Clearly, not all situations can perform a full-step — even a single automaton may create events on channels which trigger another transition indefinitely. To resolve the problem of livelock, we may ensure that there is no mutual recursion over the set of automata.

Definition 4.3.7. The *output channels* of an automaton M, written out(M), is the union of all output channels on the transitions in M. Similarly, the *input channels* of M, written in(M), are the channels appearing on the event label of transitions in M. The *dependency relation* between channels for an automaton M, written dep(M) is defined to be in$(M) \times$ out(M).
A DATE structure $\bar{M}$ is said to be *loop-free* if, for any channel c, $(c, c) \notin (\bigcup_i \text{dep}(M_i))^+$.

The following result states that only loop-free automata can perform a full-step.

Proposition 4.3.1. Given a loop-free collection of automata $\bar{M}$ in states $\bar{q}$, set of system events X and system state θ, there exist states $\bar{q}'$, system state θ' and timers T' such that $(T, \theta, \bar{q}) \Rrightarrow_X (T', \theta', \bar{q}')$.

Example 4.3.5. Considering the previous example, in practise one would like to monitor the login property for each individual user. To this end, a symbolic timed automaton does not suffice, we need a DATE which can construct a new automaton for each new user. Thus, upon a good login or bad login event, the user which generated these events is considered in the construction function and if no automaton exists for that particular user, a new automaton is constructed.

Proposition 4.3.2. An automaton in an accepting state can be removed from the DATE structure.

This clearly follows from the fact that no transition can be taken once the automaton reaches an accepting state. This implies that such an automaton can neither be triggered nor can it trigger anything.

4.4 Conclusion

Even though there are several tools for runtime verification, we felt the need to work on a logic which provides better expressivity in aspects where the existing logics are lacking. In this chapter, first we have highlighted the needs of a typical runtime verification scenario including events, conditions, actions, timers, etc. Subsequently, we have considered various design aspects of a real-time logic to specify properties of a real-time system. After making the necessary design selections, we have given the mathematical basis of the DATE logic which amongst other features uses system events, timers and channels to specify system properties.

5. Language Specification

5.1 Introduction

In the previous chapter, we have given the motivation and mathematical background of a runtime verification architecture. In this chapter, we will give a concrete language designed to capture the architecture and allow a user to express system properties. This language is referred to as LARVA which stands for Logical Automata for Runtime Verification and Analysis. In the following sections, we will gradually explain the reasons behind the design of the language and give the syntax using examples and BNF. The order of the following sections is not with respect to the language sections as they appear in a LARVA script. Rather, they are presented in an evolutionary manner, similar to the way LARVA has evolved during its creation. The basic motivations behind the syntax chosen were mainly expressivity, clarity and convenience for the user, and ease of parsing.

The next section gives the part of the LARVA syntax which can be used to describe an automaton. The syntax of events is explained in Section 5.3. This is followed by an illustrative example in the next section. The LARVA language is extended with clocks and channels in Section 5.5 and Section 5.6, respectively. Section 5.7 gives the complete BNF of the global context while Section 5.8 explains how the user can define a property for a particular context. The syntax of the invariants' and variables' sections are given in Section 5.9 and Section 5.10, respectively. Some other minor details of syntax are given in Section 5.11. To help the reader understand the syntax, Section 5.12 gives an example of a complete simple property while Section 5.13 gives an example of the language with clocks and channels. Section 5.14 concludes the chapter.

5.2 Textual Representation of an Automaton

The natural way to represent an automaton is to draw it, but a graphical user interface allowing the user to draw an automaton is beyond the scope of this project. Instead, we aim at designing an appropriate textual notation for representing an automaton — more specifically, a DATE structure.

5.2.1 Transitions

When one thinks of an automaton, usually the first thing that comes to mind are circles and arrows connecting them — transitions. A transition causes the automaton to move from one state to another. Each DATE transition has an event upon which it triggers, a condition which must be satisfied, and an action which is executed upon taking the transition. A natural way for a user to represent a transition is to use an arrow between two states. Thus a transition will be of the form: `source -> destination [ event \ condition \ action ]`. The whole set of transitions for an automaton will be given as a block enclosed with curly brackets (similar to Java) with a label *TRANSITIONS* before the beginning of the block. Consider the following example which represents an automaton monitoring whether or not a read or write in a database is done when the user is logged in:

```
TRANSITIONS
{
  loggedout -> loggedin  [ login ]
  loggedout -> loggedout [ logout ]
  loggedout -> bad_state [ read ]
  loggedout -> bad_state [ write ]
  loggedin  -> loggedout [ logout ]
  loggedin  -> loggedin  [ login ]
  loggedin  -> loggedin  [ read ]
  loggedin  -> loggedin  [ write ]
}
```

One should note that the order of the transitions is very important since the problem of non-determinism (when more than one transition can be taken) is solved by taking the first transition in the list. The BNF of transitions is given as:

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

5.2.2 States

Since the underlying DATE logic has a starting state, bad states and accepting states, we need to declare the states before actually declaring the transitions. For this purpose, a block construct similar to that of transition will be used to enclose all the states. A sub-block will be used for each of the type of states. Consider the example started above. We have a bad state (*bad_state*), a normal state (*normal*), and a starting state (*logout*):

```
STATES
{
  BAD { bad_state }

  NORMAL { loggedin }

  STARTING { loggedout }
}
```

If the user wants to execute some code upon reaching a particular state, the Java code is simply put in curly brackets following the name of the state. For example, this can be used to take a recovery action upon reaching a bad state. It can also be used for initialisation purposes by putting Java code in a starting state. However, one should note that this code is executed each time the state is entered. An example of the syntax is as follows:

```
STATES
{
  BAD { bad_state { issueWarning(); } }
}
```

The BNF for the declaration of the automaton's states is as follows:

StateDecl	::=	StateName | StateName '{' *JavaCode* '}'
StateList	::=	ϵ | StateDecl StateList
Accepting	::=	'*ACCEPTING*' '{' StateList '}'
Bad	::=	'*BAD*' '{' StateList '}'
Normal	::=	'*NORMAL*' '{' StateList '}'
Starting	::=	'*STARTING*' '{' StateList '}'
StateBlock	::=	'*STATES*' '{' Accepting Bad Normal Starting '}'

5.2.3 Properties

For better presentation, since the set of states and the set of transition together constitute the automaton, both blocks are enclosed in a single block called *Property*. The reason for this name is that an automaton represents a property about the system to be monitored. Furthermore, we would like to be able to monitor more than one property at the same time. Hence, each property is given a name. In our example the name is *accessMonitoring* as follows:

```
PROPERTY accessMonitoring
{
  STATES { ... }
  TRANSITIONS { ... }
}
```

The BNF requires the following lines to provide the definition of a property:

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

5.3 Events

Transitions require an alphabet on which they trigger. In the case of DATEs, the alphabet are events which are extracted from the target system. The basic events which we want to capture are method calls and exception throws. There are other useful system events such as variable changes, but it requires much more of an overhead to monitor variable changes. For a method call, there is a particular point where the code branches to execute the method call. On the other hand, a variable may be updated several times throughout a method execution. Thus, the overhead of monitoring variable changes is larger. To avoid this overhead and keep the language simple, we opted to stick to method calls and exception throws. We also took into consideration the fact that in Java, it is suggested that object attributes are accessed through their *get* and *set* methods. These are both detectable through method calls. Therefore, a system event declaration must include a method name. Furthermore, we would like to give a name to the event so that the user will not need to specify the method name each time the event is used. Taking these points into consideration, an event declaration would appear as follows:

```
eventName = methodCall
```

This is very limited because a lot of useful information from the method parameters is lost. The user may also need to distinguish between different methods with the same name. To meet these requirements, we add the possibility of specifying typed variables as parameters which can also be bound to the event. An event declaration now becomes:

```
eventName(Type1 arg1) = methodCall(Type1 arg1, Type2 arg2)
```

In the above example, the method call *methodCall* must have two parameters of type *Type1* and *Type2* and the event will have access to *arg1*. Methods with the same name and same argument types may appear in different classes. Moreover, a user may be interested in the object (target) on which the method is to operate. For this reason, the syntax is further extended as follows:

```
eventName(Type3 target, Type1 arg1)
        = Type3 target.methodName(Type1 arg1, Type2 arg2)
```

To continue the running example, we will declare four events: read, write, login and logout.

```
write() = {DB d.writeToDb()}
read() = {DB d.readFromDb()}
login() = {DB d.loginToDb()}
logout() = {DB d.logoutFromDb()}
```

5.3.1 Types

In the above declaration, the types of the variables sometimes appear more than once. Obliging the user to put the type repeatedly would be unreasonable. It is also not clear on which side of the declaration the type should appear. A general declaration at the start of all the events does not seem appropriate because the variables are particular to each event. Another question to consider at this point is: Should we allow the user to use the same variable name in more than one event declaration? And what if the type is different? This is resolved by allowing one variable name to be used in any event declarations within the same set of events, as long as the variable is of the same type throughout. The choice of where the type should be declared is left up to the user to decide, as long as there is at least one such declaration for each variable within a set of events. This seems to be the most straight forward option, which is both convenient for the user and avoids naming conflicts within a set of event declarations. The following declaration is a valid event declaration:

```
eventName(target, Type1 arg1, arg2, arg3)
        = Type3 target.methodName(arg1, Type2 arg2)
```

The running example can be modified to reduce the number of type declarations as follows:

```
write() = {DB d.writeToDb()}
read() = {d.readFromDb()}
login() = {d.loginToDb()}
logout() = {d.logoutFromDb()}
```

If a variable is left without a type it is simply considered as a placeholder (wildcard). The next subsection will tackle wildcards.

5.3.2 Wildcards

Sometimes it is convenient for the user to be able to use wildcards for method names. For example, if the user wants to capture all the method calls of a particular object, using the asterisk (*) this can be done as follows:

```
eventName(target) = Type target.*()
```

If we want to specify the type of the target but we are not interested in the actual object, the asterisk can be used as a placeholder. Similarly, this can be used to bind some arguments while leaving others unbound (putting an asterisk instead)[1]. The syntax is as follows:

```
eventName(Type2 arg2) = Type3 *.*(*,arg2)
```

[1]This syntax was chosen because it is very similar to AspectJ and the asterisk is used frequently as a wildcard in many applications.

Note that instead of the asterisk, the user is allowed to use any identifier with no declared type. This may sometimes be more convenient for clarity. In the running example, we can use wildcards because we do not need to bind the database (DB) object. We can also discard the type information if the method names are unique in the context. Thus, the code becomes as follows:

```
write() = {*.writeToDb()}
read() = {*.readFromDb()}
login() = {*.loginToDb()}
logout() = {*.logoutFromDb()}
```

5.3.3 Where Clause

The syntax so far is quite straight forward and self explanatory. However, this is not sufficient, sometimes we need some way of declaring custom event variables which are not directly bound to the event. For this purpose, we borrowed the keyword *where* which is commonly used in mathematics. The *where* clause can be added after each event declaration. The user can use this clause to assign any event variables which are not directly bound to the method call. This can also include any declaration of temporary variables and/or any necessary method calls. The syntax of the event declaration was adapted to be more similar to Java by introducing the curly brackets. This is also clearer for the user to see the connection between the method name and the corresponding *where* clause (especially when we introduce event collections in the next section). Furthermore, the semicolon after the method name becomes completely redundant and is therefore dropped. The syntax is as follows:

```
eventName(target, Type1 arg1, arg2, arg3)
        = {Type3 target.methodName(arg1, Type2 arg2)}
        where { Type4 arg3 = 0; }
```

Note that for convenience, the curly of the *where* clause can be left out if there is only one statement (as in Java). In the database login example, imagine we need the time at which the login occurred. The code can be modified as follows:

```
login(long time) = {*.loginToDb()} where time = System.currentTimeMillis();
```

5.3.4 Event Collections

It is convenient for the user to be able to declare collections of events. This has the advantage that an event may be triggered by more than one method call without the complications of precedence and concurrency (the event and the collection it belongs to are considered to trigger simultaneously with no precedence over each other). Another considerable advantage is that the *where* clause specified for the collection is automatically applied to all the events in the collection. In the declaration of collections the pipe symbol (|) is used because in other well known notations (e.g. BNF) this represents choice. The syntax is as follows:

```
eventCollection() = { event1 | event2 | ... }
```

The running example can be modified to include a definition for event *access* which triggers when either a read or a write occurs:

```
access() = { write | read }
```

The sub-events need not be previously declared events. Therefore, in an event collection the user can still declare primitive events from method calls. This may be more convenient for the user by avoiding to declare events which will never be used on their own (but rather as part of a collection). Another notable advantage is that each stand-alone event requires that its *where* clause initialises all the necessary variables (specifically those which are not bound to the method). As noted earlier the *where* clause of a collection is applied to all the sub-events. Therefore, allowing the user to declare the primitive events in a collection will avoid a lot of unnecessary duplicated *where* clauses. The syntax is as follows:

```
collection() = { {methodName()} where { ... } | declaredEvent }
```

Consider the scenario where we need the time at which the database was accessed. This can be defined directly on the collection as follows:

```
access(long time) = { write | read } where time = System.currentTimeMillis();
```

Parameters

A question arises from the fact that different events in the collection have different parameters: Which parameters will be allowed to be accessed by the collection? To resolve this dilemma one must recall that we are allowed to declare any variable in an event as long as it is either directly bound to the method or initialised in the *where* clause. Therefore, it is reasonable to keep the same policy in the event collection. This means that any variable which is not bound by all the collection's sub-events, must be initialized in the collection's *where* clause. Consider the following event declarations:

```
event1(Type1 arg1, Type2 arg2)
      = {Type1 arg1.methodName1(Type2 arg2)}

collection(Type1 arg1, Type2 arg2, Type3 arg3)
      = { {Type1 arg1.methodName2()} where { arg2 = 2; } | event1 }
      where { arg3 = 0; } }
```

In this example, *event1* does not bind *arg3*. Therefore, it must be provided for in the *where* clause of the collection. The same thing happens with *methodName2*. The database login example can be modified to keep track of the name of the user which logged in (if this is available as a parameter). The code can be modified as follows:

```
login(String name) = {*.loginToDb(String name)}
```

Where Clause

Repeatedly, we have reaffirmed that the collection's *where* clause is applied to all the sub-events. The problem here is: What will happen if the initialisation of a variable is provided twice for a particular event? It will be very misleading if the initialisation of the collection's *where* clause overrides that of the individual event (previously declared). The solution which seems to be most feasible is that the *where* clause which is most specific to the event declaration is never overridden later on. To make this clear and avoid confusion, the parser of the language issues a warning that a part of the *where* clause of the collection is being ignored for a particular event (because it has been already initialised in a more specific *where* clause). Consider the following example:

```
event1(Type1 arg1, Type2 arg2)
      = {Type1 arg1.methodName1(Type2 arg2)}

collection(Type1 arg1, Type2 arg2, Type3 arg3)
      = { {Type1 arg1.methodName2()}
      where { arg3 = 3; } | event1 }
      where { arg2 = 2; arg3 = 0; }
```

In this example, an interesting conflict emerges: *arg2* for *methodName1* is initialised both by the method call and also by the *where* clause collection. Similarly, *arg3* for *methodName2* is initialised in both *where* clauses. In order to solve such conflicts of duplicated initialisation we always apply the *where* clause which is most specific to the event in question. This means that the initialisation *arg2=2* will not be applied on *methodName1* while the initialisation *arg3=0* will not be applied on *methodName2*.

Using the database example, we can show how a *log* event can be created using the *login* and *logout* events. In this case we want to pass a meaningful string depending on the event being logged. Thus different *where* clauses have to be defined for some sub-collections. A default message may be applied to all the events with no specific clause. The following code shows how a specific string is available for the event *login* but for the event *logout* the default message is applied:

```
login(String name) = {*.loginToDb(String name)}
logevent(String info) = { {login} where {info = name;} | logout }
                        where { info = "default message"; }
```

Sub-collections

Purely for the sake of convenience, the user is also allowed to declare sub-collections in collections. In fact, there is no limit to the depth of the nesting of event collections. The syntax is as follows:

```
collection() = { ev1 | { ev2 | ev3 } where { ... } }
             where { ... }
```

Based on the running example, we can construct the following to group all the events together:

```
access() = { write | read }
anyEvent() = { access | { login | logout} }
```

5.3.5 Event Variations

Having only a one-to-one mapping between events and method calls is too restrictive. For example, the user may need to verify something before the method is called and after it returns. Therefore, we need to allow the user to specify different types of events for the same method call. We identified four different variations of events for the same method call. These are:

(i) before the method execution, written as:

```
{methodName()}
```

(ii) after the method returns, written as:

```
{methodName() uponReturning (returnedObject)}
```

(iii) after the method throws an exception, written as:

```
{methodName() uponThrowing (thrownException)}
```

(iv) upon the handling of an exception — the start of a *catch block*, written as:

```
{methodName() uponHandling (handledException)}
```

In each of the last three event types, the user can also bind to the returned object or the thrown/handled exception accordingly.

Considering the running example, we would like to know whether the *login* method had to handle an exception during its execution, or possibly thrown an exception. Thus, we can define the two occurrences as a collection:

```
loginException(Exception ex) = { *.loginToDb() uponThrowing (ex)
                               | *.loginToDb() uponHandling (ex) }
```

5.3.6 BNF

Below, we give the BNF of the *Events* section which is the most complex part of the language:

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 ‘}’

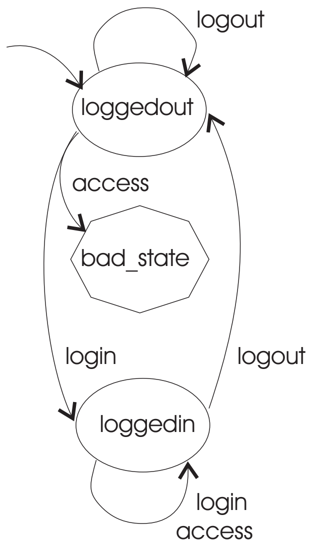
5.4 Database Access Example

To summarise the previous section and illustrate the running example which we have used all along, we present the complete script together with an illustration of the automaton in Figure 5.1. Note that by introducing the event collection *access* we have removed two transitions from the original list presented in Subsection 5.2.1.

5.5 Clocks

Clocks can be used in a Larva script to define real-time properties. For example with clocks we can define that no more than three bad logins should occur within five minutes. Another example would be to specify that a transaction which failed should be retried within a maximum of ten minutes. There are also a lot of real-time applications which are related to safety critical operations. For example it should be ensured that the train gate is closed within five seconds after the closing signal was sent. Such a simple check may prevent a possible tragedy.

A clock may appear in various parts of a Larva script because according to the DATE logic it can generate events, it can be used as part of conditions, and it also supports actions. However, it cannot be treated like the other variables used in the declaration of events because a clock may appear in the transitions without appearing in the events section. Furthermore, a clock cannot be treated as a



```
GLOBAL
{
  EVENTS
  {
    write()  = {*.writeToDb()}
    read()   = {*.readFromDb()}
    login()  = {*.loginToDb()}
    logout() = {*.logoutFromDb()}
    access() = { write | read }
  }

  PROPERTY accessMonitoring
  {
    STATES
    {
      BAD { bad_state }
      NORMAL { loggedin }
      STARTING { loggedout }
    }

    TRANSITIONS
    {
      loggedout -> loggedin  [ login ]
      loggedout -> loggedout [ logout ]
      loggedout -> bad_state [ access ]
      loggedin  -> loggedout [ logout ]
      loggedin  -> loggedin  [ login ]
      loggedin  -> loggedin  [ access ]
    }
  }
}
```

Figure 5.1: The automaton and Larva code of the database access example.

parameter of a method — it is part of the clock configuration of the automaton. Thus, a clock should be declared before the events section. This will be done in a section called *Variables* as follows:

```
VARIABLES
{
  Clock c;
}
```

The BNF of the *Variables* section is as follows:

ClockName	::=	identifier
ClockDecl	::=	'*Clock*' ClockName ';'
VariableDecl	::=	ClockDecl
VariableList	::=	VariableList VariableDecl | ϵ
VariablesBlock	::=	'*VARIABLES*' '{' VariableList '}'

A simple way to declare events based on a clock, is to state the clock and the time at which it should trigger separated by the symbol @. A clock event triggering four seconds after the last clock reset is declared as follows:

```
clockEvent = {c@4}
```

In practical cases encountered, it is sometimes useful to have a clock which triggers repeatedly after a number of seconds. Instead of resetting the clock after each event, the clock can be set to trigger for every second which is a multiple of a particular number. Therefore, one can also declare *c@%4* signifying that the clock event will occur repreatedly after every 4 seconds.

The BNF of events will be modified as follows:

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

A number of methods are available with the *Clock* and can be used just like normal methods wherever Java code is allowed in the LARVA script. These are intended to make clocks more usable and versatile. For example: *clockName.reset()* will cause the clock *clockName* to be reset.

- The *reset* method is the one which should be used whenever one requires to restart the clock. Initially, the clock is automatically started as soon as its context starts existing. This means that as soon as the monitoring system starts up, any clock declared in the *Global* context is started. Similarly, upon the start of a new automaton for a particular context, all the clocks in that context will be automatically started.
- The *current* method will return a double with the number of seconds elapsed since the clock was started/reset.
- The *compareTo* method accepts a double as a parameter and returns an integer. If the integer is zero, then the current clock value and the parameter are equal. If the integer is positive then the clock value is larger than the parameter. Otherwise, the clock value is smaller than the parameter.
- A clock can be switched off using the method *off*. This can be especially useful for clocks which are set to generate an event repeatedly for a certain amount of time.
- A clock can be paused using the method *pause*.
- A clock can be resumed using the method *resume*.

5.6 Channels

The DATE logic supports channels for automata communication. For example, one would like to start an automaton upon the completion of another. Or one would like to trigger a transition when another automaton reaches a particular state. Channels in DATEs are broadcast channels which means that all the automata will receive the channel-triggered events. In the future this can be improved either by adding a tag to the channel event — so that events are distinguishable — or by introducing point-to-point channel communication.

Declaring a channel is very similar to declaring a clock — first it should be declared in the variables section — since it is not related to any particular system event. The declaration is as follows:

```
VARIABLES
{
  Channel d;
}
```

In order to declare an event upon the channel, the question mark symbol (?) is used because this is similar to the way other languages (such as CSP and Occam) represent channel communication. Objects can also be sent and received through the channel. A code example is shown below:

```
EVENTS
{
  channelD(Object obj) = {d?obj}
}
```

For the same reason, the exclamation mark (!) is used to represent sending an event over a channel. A trigger of a channel event is considered as an action and thus, it is found as part of an action on a transition. Objects can be sent through the channel as shown in the example below:

```
TRANSITIONS
{
  starting -> normal [an_event\\d!obj]
}
```

The BNF of the variables and events will be modified as follows:

ChannelName	::=	identifier
ChannelDecl	::=	'*Channel*' ChannelName ';'
VariableDecl	::=	ClockDecl | ChannelDecl
VariableList	::=	VariableList VariableDecl | ϵ
ChannelEvent	::=	ChannelName '?' identifier
PrimitiveEvents	::=	PrimitiveEvent | ClockEvent | ChannelEvent

5.7 The Global Context

All the sections described so far can be put in one container which will be called the *global context*. A global context has a number of clocks and channels in the *Variables* section, a number of events and a number of properties (automata). The complete BNF for the global context is as follows:

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

5.8 Context

In order to specify a property for each object of a particular type, we introduce the *FOREACH* construct. For example, all the invariants, variables, and properties within a *FOREACH (User u)* section will be applied for each instance of type *User*. It is important to note that each event with a *FOREACH* must provide a context by assigning a value to the context variable using the *where* clause. The syntax is as follows:

```
FOREACH (User u)
{
  INVARIANTS { ... }

  VARIABLES { ... }

  EVENTS
  {
   addUser() = {*.addUser(User u1)} where { u = u1; }
   ...
  }

  PROPERTY { ... }
}
```

The context of a property may be made up of more than one variable. For example, a property may apply *foreach* user and *foreach* account. The set of possible automata will be the cross product of all the users and all the accounts. Sometimes, this approach restricts the user. Consider a scenario where we want to declare a property about all the users, and another property about all the accounts (in the context of their users). In the first property we do not need the context of an account because the property concerns the user but not the accounts. Placing this property in a context of a user and an account clearly provides too much context. The solution is to allow a *Foreach* section to contain other *Foreach* sections. This is referred to as *nesting*.

5.8.1 Nesting

Nesting allows the user to specify *exactly* the necessary level of context. The properties which concern the user without the account, are put in the *user* context while the properties which concern both the user and the account are placed in a more specific context. For example, for each user, we would like to ensure that there are no more than five pending requests, and we may want to place an invariant restriction on the ID of each account. For the former property the user context is sufficient. However, the second property has to be applied for each account of each user. Note that, each event must provide a value to the *whole* context even if the context is nested. This is shown in the following code snippet:

```
FOREACH (User u)
{
  ...
  FOREACH (Account a)
  {
    EVENTS
    {
      addAccount() = {User u1.addAccount(Account a1)}
                     where { a = a1; u = u1; }
    }
    ...
  }
}
```

The BNF of contextual properties is as follows:

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

Note that the definition of *Context* has been modified to include *SubContext*.

Nesting of contexts creates the issue of referring to variables in higher-level contexts. For example, we are monitoring the number of users using the variable *count* and monitoring the number of accounts of each user using another variable *count*. Although the variable names are the same, they belong to different contexts. Recall that the count of users is still available in the account context. To resolve this conflict and make it clear to which variable in which context we are referring to, we allow the user to insert special notation in the Java statements used in LARVA. This is achieved using the symbol "::" together with the name of the context as follows:

```
FOREACH (User u)
{
  VARIABLES { int count = 0; }
  ...
  FOREACH (Account a)
  {
    VARIABLES { int count = 0; }

    PROPERTY
    {
     ...
     if (u::count > 5 || u::a::count < 6)
     ...
    }
  }
}
```

5.8.2 Context and Clocks

The clocks in Larva can be contextual. This means that if the clock is declared in a *Foreach* context, then a separate clock will be created for each monitored object. On the other hand, if it is declared in the *Global* context, only one clock is created and it will be available for all the contexts and automata. This means that even the generated events will be broadcast to all the sub-contexts (unlike the other events).

5.8.3 Context and Channels

Channels in Larva are never contextual — they are always global. In other words, they are broadcast channels whose events are received by all the automata. As future work, we can also introduce point-to-point channels. As a compromise, the channel specification (above) allows tags to be sent over channels. Using these tags, the receiving end can distinguish between senders. Note that this has been implemented in the current Larva implementation and used in the last case study. However, the notion of tags has not been yet included in the mathematical framework of channels.

5.9 Invariants

Invariants are aspects of the system state which we would like to remain unchanged. For example we would like the status of the system to remain unchanged during the verification of a particular property. We need three basic parts for such a declaration: the Java statement to obtain the status, the name of the invariant and the type returned by the Java statement.

The following example shows how the invariant *sysStatus* is declared:

```
INVARIANTS
{
  int sysStatus = System.getStatus();
}
```

Invariants can appear both in the global context and in the sub-context. If it is in a sub-context it will be applied to all the replicated automata of that context. Thus the BNF definition of *context* becomes as follows:

Type	::=	identifier
VariableName	::=	identifier
Invariant	::=	Type VariableName '=' identifier '.' MethodName '(' ')'
Invariants	::=	Invariants Invariant | ϵ
InvariantsBlock	::=	'*INVARIANTS*' '{' Invariants '}'
Context	::=	InvariantsBlock VariablesBlock EventsBlock PropertyBlocks SubContexts

5.10 Variables

So far the only declarations in the variables section were of type clock and channel. However, any type of variable may be required, especially integer variables to be used as counters in the automata. The variables declared in the *Variables* section can be considered as the local variables of the automaton. The variable declarations allowed are the same as variable declarations in Java. For example:

```
VARIABLES
{
  int i = 0;
  boolean b = false;
}
```

The complete BNF of the *Variables* section is as follows:

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

5.10.1 Initialisation

An interesting question which crops up is where we should place the initialisation of the variables. We considered two possible approaches: provide a specific place where the initialisations are declared, or associate the initialisation with the action of the starting node of the automaton. Associating the initialisation code with a starting node may be confusing — what happens if the automaton re-enters the starting state? Will the initialisation code run again? Therefore, instead of this option, we chose to allow the user to perform initialisation upon declaration of the variable as in Java. Still, the drawback may be that we want to re-initialise the automaton to the starting state at some point in the execution. Or maybe the user needs to perform a complicated initialisation. For these reasons, in the future we will consider re-designing the issue of initialisation altogether.

5.11 Other Details

5.11.1 Imports

A script written in Larva does not have any context of Java packages Therefore, it is mandatory for the user to provide the necessary classes as imports in the *imports* section of the Larva script. Any object that is used in the script but which is not part of the standard Java must be included as an import. The same rules of a Java import section apply to the Larva import section. The syntax is given below:

```
IMPORTS
{
  import a_package.sub_package.ClassName;
  ...
}
```

5.11.2 Methods

For the specification of properties, sometimes various line of code may be required for a transition's action. This also applies to wherever there is Java code in the Larva script. For this reason, the user is allowed to declare these statement as a method and thus, the user can simply use the method name instead of all the lines of code. The syntax is as shown below:

```
METHODS
{
  boolean checkNumber (int number)
  {
    return true;
  }
}
```

5.12 Illustrative Example: Bank System

To illustrate the use of Larva, consider the monitoring of a simplified banking system. We want to monitor that there should never be more than five users in the bank and that a deletion does not occur when there are no users. For this purpose, we specify an automaton whose transitions trigger upon system events. This automaton will have various types of states such as a starting state and some bad states which represent the violation of a property. The next step would be to identify the system events corresponding to the method calls in the target system. Upon the occurrence of particular events the automaton transitions trigger, updating the state of the system. To keep track of the number of users it is we will use a discrete variable which can be accessed and manipulated by the automaton through actions on the transitions. For example, upon the method call *addUser* we increment the variable as shown in the first transition of the code in Figure 5.2. To check whether the number of allowed users has been exceeded, we need to check certain conditions upon a transition. Transitions may only trigger if an associated condition holds. Therefore, each transition will have three parts: an event, a condition and an action. In Figure 5.2 we show the automaton used for monitoring the addition and deletion of users, together with the equivalent LARVA code[2].

5.13 Example with Clocks and Channels

In this section, we will give a small example of how the clocks and channels can be used. First we describe a realistic scenario and we show how the property can be expressed as a DATE.

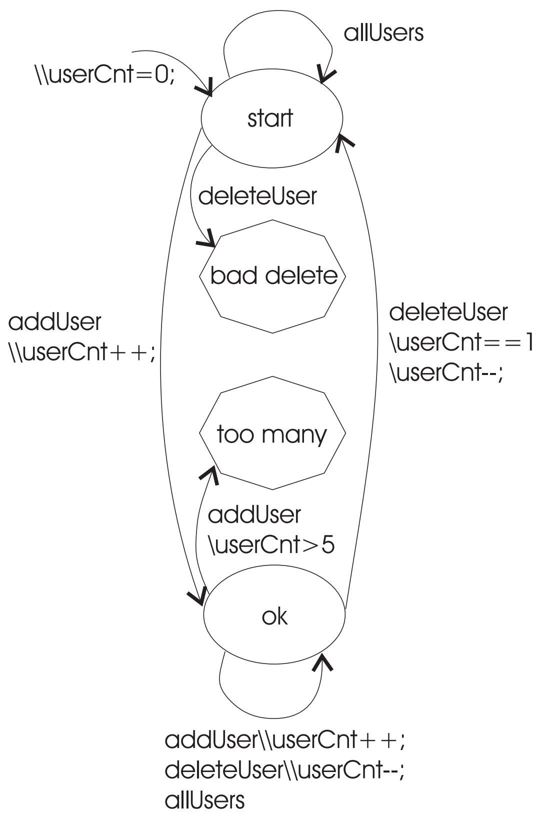
A server should not be overloaded with traffic such that users are denied service. If the traffic is high for a long period of time, the probability of a denial of service is high. Thus, the server is monitored to check for long periods of high volumes of traffic. If the traffic is repeatedly found to be high over a period of time, then the frequency of user denial of service is checked. If a denial of service occurs frequently, then the administrator is notified of the problem.

In the script that follows, there are two automata. The first automaton *highTraffic* triggers regularly upon a timer event. If the server is handling high traffic, a counter is incremented, otherwise it is decremented. Once the counter reaches 5, an event is sent over channel *d* to automaton *denial*. Similarly, the automaton keeps count of the number of denial of service occurrences. If the count exceeds 5 the automaton reaches bad state *problem*.

```
GLOBAL
{
  VARIABLES
  {
    Clock c = new Clock();
    Channel d = new Channel();
    int count = 0;
  }

  EVENTS
```

[2]For a complete user manual of LARVA and further examples please refer to `http://www.cs.um.edu.mt/~svrg/Tools/LARVA`. The tool is also available for download.



```
GLOBAL
{
  VARIABLES
  {
    int userCnt = 0;
  }

  EVENTS
  {
    addUser() = {*.addUser()}
    deleteUser() = {*.deleteUser()}
    allUsers() = {User u.*()}
  }

  PROPERTY users
  {
    STATES
    {
      BAD { too_many bad_delete }
      NORMAL { ok }
      STARTING { start }
    }

    TRANSITIONS
    {
      start -> ok [addUser\\userCnt++;]
      start -> bad_delete [deleteUser\\]
      ...
      ok -> ok [deleteUser\\userCnt--;]
      ok -> ok [allUsers]
    }
  }
}
```

Figure 5.2: The automaton and Larva code of a hypothetical bank system.

```
{
  clockC() = {c@10}
  channelD() = {d?}
}

PROPERTY highTraffic
{
  STATES
  {
    STARTING { starting }
  }

  TRANSITIONS
  {
    starting -> starting
```

```
        [clockC\system.highTraffic() && count > 5\c.reset();d!]
      starting -> starting
        [clockC\system.highTraffic()\c.reset();count++;]
      starting -> starting
        [clockC\count>0\c.reset();count--;]
      starting -> starting
        [clockC\\c.reset();]
    }
  }

  PROPERTY denial
  {
    STATES
    {
      BAD { problem }

      NORMAL { normal }

      STARTING { starting }
    }

    TRANSITIONS
    {
      starting -> normal [channelD]
      normal -> problem [userDenied\count > 5]
      normal -> normal [userDenied\\count++;]
      normal -> normal [clockC\count > 0\c.reset();count--;]
      normal -> normal [clockC\\c.reset();]
    }
  }
}
```

5.14 Conclusion

In this chapter we have provided a suitable language which is relatively easy to use and parse based on the DATE logic presented in the previous chapter. The language has evolved through experimentation in practical and even real-life scenarios. For this reason, the language has been presented as an evolving language — starting from a basic symbolic automaton to the full Larva automaton. Two examples have been given to illustrate the use of the language. In the next chapter we will give the complete view of the proposed runtime verification architecture, together with the implementation details involved.

6. Larva Implementation Details

6.1 Introduction

In this chapter, we will explain the implementation details of the Larva architecture. First, in Section 6.2, we explain the basic setting on which runtime verification can be applied. In Section 6.3, we introduce aspect-oriented programming because of its key role in the monitoring system by injecting monitoring code in the target system. Subsequently, the architecture of Larva — the way it is integrated with the monitored system — is explained in Section 6.4. In the following section, we explain the implementation of the Larva automaton in Java. This includes the hierarchy of classes which implement the monitoring system and how the context of each automaton is stored. Another interesting detail is the implementation of invariants which monitor the changes in attributes of objects. This is explained in Section 6.6. A very challenging part of the implementation is that concerning real-time. This is mainly because the standard Java virtual machine does not strictly support real-time. The details are in Section 6.7. This is followed by a section explaining another difficult implementation aspect — that of channels. The last section concludes the chapter.

6.2 Composing a Larva Script

Larva is a system which implements DATE structures, allowing the user to specify several properties to be verified during the execution of a system. These properties are automatically translated into the necessary Java code using the Larva compiler. The code will include two main components: the code which extracts the events from the underlying system, and the code which verifies the properties based on the generated events.

To summarize and illustrate the whole process of using Larva, we will use block diagrams together with an example. In order to be able to verify a system, we obviously need a system and its specification written in Larva. The Larva specification includes the system events and the properties to be verified.[1] Figure

[1]More information about the Larva language can be found in the Larva manual provided

6.1 shows the components which must be provided by the user.

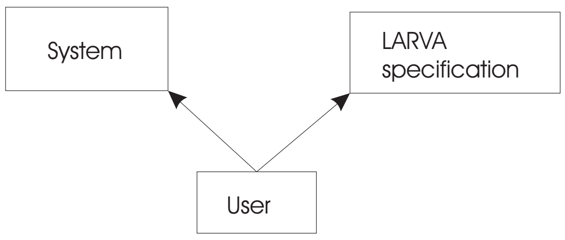


Figure 6.1: The components which must be provided by the user.

As an example we will consider a logging system which ensures that a read and write operations are only allowed when the user is logged in. In this case, the system includes the Java classes which implement the logic. An idea of what the system code might look like is provided in the following code snippet:

```
public class SystemMain
{
  public void loginToDb() { ... }

  public void logoutFromDb() { ... }

  public void writeToDb() { ... }

  public void readFromDb() { ... }
}
```

Given a system to be monitored, the user must provide the LARVA specification, which comprises the list of events and properties to be verified. An example of a script related to the logging system is the following:

```
  ...
  EVENTS
  {
    write() = {*.writeToDb()}
    read() = {*.readFromDb()}
    login() = {*.loginToDb()}
    logout() = {*.logoutFromDb()}
  }
  ...
  TRANSITIONS
  {
    start -> start [ login \\ loggedin = true; ]
    start -> start [ logout \\ loggedin = false; ]
    start -> bad_write [ write \ !loggedin \ ]
    start -> bad_read [ read \ !loggedin ]
  }
  ...
```

online: http://www.cs.um.edu.mt/svrg/Tools/LARVA/.

Before this script is of any use, it must compiled by the Larva compiler to generate the equivalent automaton implementation in terms of Java and AspectJ code. AspectJ is an aspect-oriented programming language which complements Java. The following section will give a brief overview of aspect-oriented programming to help the reader understand how the runtime verification architecture works.

6.3 Aspect-Oriented Programming

Larva uses aspect-oriented programming techniques [62] to capture events. More specifically, AspectJ technology was chosen because of the available support and tools. The advantage of using aspect-oriented programming is that *crosscutting* concerns can be defined as a separate module. For example, consider accounting concerns: code which counts the number of resources used will be scattered wherever there is a resource acquisition or a resource release. Being able to capture the required logic as one module is very convenient: it is faster to implement and less error-prone. Aspect-oriented programming is able to insert code in specific points in the system code based on matching rules. Such specific points — also known as *joinpoints* — are identifiable points in the code. Examples include method calls, exception throws and so on. In an aspect file, one can define a *pointcut* which is a sort of rule to which a number of joinpoints should match. An *advice* is a piece of code associated with a pointcut which is executed in the place where a joinpoint matches the corresponding pointcut. The insertion of advice code is called *instrumentation* or *weaving.* This can occur at compile-time or at load-time. Compile-time weaving requires the source code of the system being weaved, while the load-time weaving works on the byte-code. The former is faster but sometimes one may prefer not to recompile the whole system. Another advantage of load-time weaving is that is that the system can be run with or without aspects with more flexibility.

As an example of how aspect-oriented programming works consider a system which creates shapes and displays them on screen. Using the usual techniques, keeping a count of all the shapes currently displayed, one must keep some kind of global variable which is updated by all the objects in the system. Using aspect-oriented programming, this task becomes much simpler. Consider the following code for the *Circle* object:

```
public class Circle implements Shape
{
  public void display() { ... }

  public void hide() { ... }
}
```

Similar code exists for the objects *Triangle*, *Square*, and *Rectangle.* In the example, *Shape* is an interface and hence cannot have any implementation. Furthermore, keeping count of the shapes is not considered part of the system functionality. The solution given by aspect-oriented programming involves the following few lines of code:

```
public aspect Shapes
{
  int count = 0;

  before():(execution(* *.display())) { count ++; }

  before():(execution(* *.hide())) { count --; }
}
```

The actual semantics of the code upon weaving would be equivalent to the following code:

```
public class Circle implements Shape
{
  public void display() { count ++; ... }

  public void hide() { count --; ... }
}
```

Note how the statements in the pieces of advice, have been inserted in the code of the system at the specified joinpoints. Aspect-oriented programming is very useful for runtime verification. Through this technology, code which checks the state of the system can be inserted in the required parts without the need to actually alter the code. This makes it both easier and less error-prone. More importantly, the weaving is done automatically with no effort from the part of the runtime verification designer.

In the case of Larva, the automaton which monitors the system, is given control upon the occurrence of specific events which are relevant to the monitoring. Subsequently, the automaton performs the necessary operations and passes control back to the system. The necessary operations include the initialisation of automata, logging of information, update of the state of automata, and taking appropriate actions to remedy particular situations as specified by the user.

6.4 Larva for Runtime Verification

In the previous two sections, we have first laid out the setting of a typical system which will be augmented with runtime monitoring, and then, we introduced aspect-oriented programming. In this section, we will show how the Larva script introduced in Section 6.2 is compiled in order to generate the necessary Java and AspectJ code. The input and outputs of the compilation process are shown in Figure 6.2.

Building on the example, the Larva code (a part of which is shown above) is compiled and the Larva compiler generates an aspect file which captures the

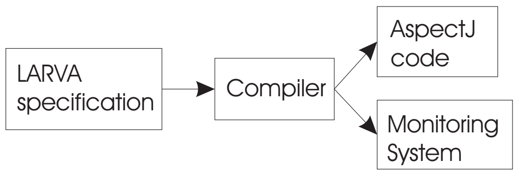


Figure 6.2: Generating the Java code using the compiler.

events, and a class file with the logic of the automaton. Upon capturing an event, the code in the aspect triggers the automaton to perform relevant steps. A snippet of the aspect code is the following:

```
before () : (call(* *.loginToDb(..)) && !cflow(adviceexecution()))
{
  _cls_property _cls_inst = _cls_property._get_cls_property_inst();
  _cls_inst._call(thisJoinPoint.getSignature().toString(), 4 /*in*/);
}
```

Note that the method *_call* implements the automaton logic through a series of if-then-else statements. A snippet of the automaton code is the following:

```
...
else if (_state_id_badAccess == 2 /*start state*/)
{
  if (1==0) {}
  else if (_occurredEvent(_event,4 /*in*/))
  {
    loggedin = true;
    _state_id_badAccess = 2; //moving to start state
    _goto_badAccess(_info);
  }
  ...
```

Note that each state and each event are given an identification number. The above code is checking that if the automaton is in the *start* state and the event *in* occurs, then, the variable *logged* is set to true and the automaton goes again to the *start* state. This is in line with the LARVA definition of the transitions. The last method call *_goto_badAccess* saves some useful logging information for the user.

Once the user has the LARVA compiler-generated code, this is recompiled together with the system code using the AspectJ compiler. The latter automatically weaves the monitoring and verifying code with the system code. The end configuration will be like the one shown in Figure 6.3.

When the monitoring code is weaved with the system code, the system is in fact modified with additional code in the strategic places indicated by the aspect code. Considering the above two code snippets, the monitoring code will be injected exactly *before* the *call* to the method *loginToDb* indicated in the first code snippet. This mechanism will update the automaton according to the

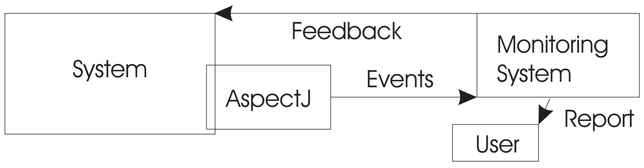


Figure 6.3: Larva in practise.

system events taking place. In case a login occurs, the variable *logged* is set to true and the automaton performs the steps as described above. Similarly, if the method *readFromDb* is called and the variable *logged* is false, the monitoring system immediately reaches a bad state before the method *readFromDb* is actually invoked. Using the same technique, a report of the monitoring situation (including the states through which the system is going through and any bad states encountered) is issued in a text file (see Figure 6.3). This information also provides the stack trace where the bad state was encountered.

6.5 Implementation of the Automaton

The implementation of the automaton in Java is very straight forward: it is implemented as a series of nested if-then-else statements which represent the selection of the source location (according to the current state of the automaton) and the selection of the destination location (according to the current events and variables). The if-then-else construct also allows us to easily implement the priority of the transitions. Recall that for the sake of determinism, transitions are ordered by priority. A simple integer variable is used to keep track of the current state of the automaton. The more complex part of the implementation involves automata with contexts. A context refers to the replication of automata according to their uniqueness. The following subsections explain the structure used to represent contexts, how we keep track of the different automata, and how the uniqueness of the objects is decided.

6.5.1 Structure of Contexts

In order to keep track of a context we have to associate an automaton with the object (context) being monitored by that automaton. In this way the automaton itself inherits the uniqueness of the object — this is what distinguishes it from the other automata. Another important point is that a context should have access to the variables of all the higher-level contexts. For example a user context, which keeps track of the number of accounts, should also have access to the global context (above the user context) which keeps track of the number of users. For this reason, we require that, at least, each context has access to its immediate parent context. Furthermore, the uniqueness of a context also depends on the

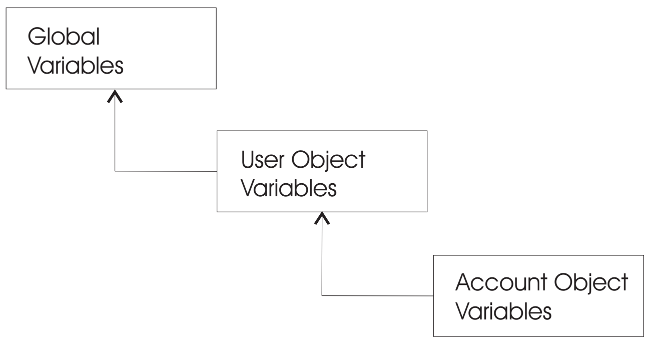


Figure 6.4: The implementation structure of contexts.

uniqueness of its parent. Therefore, this should also be included in the *equals* method which decides the uniqueness of the automaton.

In summary, the basic references required for a context include: (i) a reference to its immediate parent (and thus a reference to the parent's context and variables); (ii) a reference to the objects which make up the context of the automaton; and (iii) a reference to the local variables. As an example consider a global context (which monitors the number of users), a user context (which monitors the number of accounts each user has), and an account context (which monitors the transactions in each account). This is shown in Figure 6.4.

6.5.2 Dynamic Triggers

To monitor properties with context, we must trigger a new automaton for each unique object encountered. For this purpose, a hash table is used to store the objects being monitored. Upon receiving an event, the monitoring system checks whether the object, responsible for the event, is already in the hash table; if not, a new automaton is created and initialised. To use the same example, consider a scenario where a login/logout is done on a per user basis. The following code snippet illustrates what we have been saying:

```
public static HashMap <_cls_property, _cls_property>
  _cls_property_instances = new HashMap <_cls_property, _cls_property>();
...
public static _cls_property _get_cls_property_inst (User u)
{
  _cls_property _inst = new _cls_property(u);
  if (_cls_property_instances.containsKey(_inst))
  {
    _cls_property tmp = _cls_property_instances.get(_inst);
    return _cls_property_instances.get(_inst);
  }
  else
  {
    _cls_property_instances.put(_inst,_inst);
    return _inst;
  }
}
```

In the above code, a *HashMap* is used to store all the active instances of a particular automaton. Both the key and the value of the *HashMap* are the automaton itself because the automaton object has an *equals* method which is able to distinguish automata from each other. The static method *_get_cls_property_inst* in the code, accepts a context (in the example, a *User* object) as input and returns its corresponding automaton. First, the method tries to find the automaton in the *HashMap*; if not found, it creates a new one and puts it in the *HashMap*. This code is called each time an event is received so that the event is applied to the appropriate automaton. Recall that each event is related to a context through its *where* clause. The actual code called before the event *loginToDb* is the following:

```
before (User u1) : (call(* *.loginToDb(..)) && args(u1)
  && !cflow(adviceexecution()))
{
  User u;
  u = u1;
  _cls_channelexample1 _cls_inst
     = _cls_channelexample1._get_cls_channelexample1_inst(u);
  _cls_inst.u1 = u1;
  _cls_inst._call(thisJoinPoint.getSignature().toString(), 4 /*in*/);
}
```

Note how the context variable *u* is first obtained from the method parameter, and then passed as a parameter to obtain the corresponding automaton. Unfortunately, this approach requires the storage of all the monitored objects. The consequence may be a considerable overhead on the system's resources. The solution for this problem is to use accepting states. Upon reaching an accepting state, an automaton is considered to have reached its purpose and can thus be removed from memory. Recall that in the definition of DATEs (Section 4.3) an accepting state cannot have any outgoing states. Thus, once the automaton reaches such a state, it can be discarded.

In the case of nested contexts, a part of the context passed to the creation/retrieval of an automaton is actually used to create/retrieve the parent context. Recall that a context has reference to its parent's context. The following code shows the process of creating an automaton with a nested context (transaction context within a user context):

```
public _cls_property (Transaction t, User u)
{
  parent = _cls_property._get_cls_property_inst(u);
  this.t = t;
}
```

Note that the first line of code in the method, retrieves a pointer to the parent automaton. The subsequent line of code sets the context of the current automaton. Two identical contexts can still be distinguishable because of different parents. The code for comparing two automata with a nested context is as follows:

```
public boolean equals (Object o)
{
  if ((o instanceof _cls_property)
    && (context == null || t.equals(((_cls_property)o).context))
    && (parent == null || parent.equals(((_cls_property)o).parent)))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

6.5.3 Equality of Objects

An interesting implementation issue is the problem of comparing objects, i.e. which objects should be considered to be the same object. This is very important when monitoring properties according to a context. For example, consider monitoring a property for transactions which are serialised and deserialised during their life cycle because of communication. From a practical perspective, these should be considered as the same object even though they are not the same instance. For this reason, we do not construct automata upon the construction of objects. Rather, Larva uses the *equals* method of objects in order to decide their uniqueness. Thus, if two objects are not equal, they should each have a separate automaton. If the default *equals* method is not appropriate for the purpose of runtime verification, the user can implement a custom *equals* method in the Larva script.

The following example will be used to highlight the need for this flexibility. Consider a transaction object which is created without an identification but which is given this identification at a particular stage in its life cycle. Furthermore, this object is serialized and sent over a communication channel. Subsequently, the object is received and deserialised. Also assume that the transaction object does not have an implemented equals method (it will inherit the equals method of *Object*). The only solution is to provide a custom equals method in the Larva script. A possible equals method is the following:

```
public boolean compareTransactions (Transaction t1, Transaction t2)
{
  if (t1.getID() == null)
    return t1.equals(t2);
  else
    return t1.getID().equals(t2.getID());
}
```

Note that the custom equals methods requires two instances of the the compared object as inputs. To notify LARVA to use this method, instead of the default *equals* method, the *equateUsing* construct should be used as follows:

```
...
FOREACH (Transaction t equateUsing compareTransactions)
...
```

The actual comparison of the context is done automatically by the *HashMap* mechanism. For this reason, we have used the automaton object as the key of the *HashMap*. Before adding any objects to the *HashMap* we check whether the *HashMap* already contains the key using the *containsKey* method. This automatically invokes the *equals* method of the automaton object. Automaton objects are compared according to the objects which constitute their context. Recall that if the user does not specify an *equals* method, the comparison is done using the default *equals* method. On the other hand, if the user specifies a comparison method, this is used instead. The following code shows a context which is made up of a *Transaction* and a *User*. The former is compared using the standard default *equals* method, for the latter the user-specified method *comparer* is used:

```
public boolean equals (Object o)
{
  if ((o instanceof _cls_property)
    && ( transaction == null || t.equals(((_cls_property)o).transaction))
    && ( comparer(user, ((_cls_property)o).user))
    && (parent == null || parent.equals(((_cls_property)o).parent)))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

6.6 Invariants

Recall that invariants are attributes of an object which should not change throughout a number of transitions. In order to keep track of the violation of invariants, a variable is used to store the last known value of the invariant attribute. Before taking a transition, the stored value is compared to the current value of the entity. If they match, the transition is taken as usual; if a discrepancy is found, the automaton reverts to a bad state. In mathematical terms this means that we have added another condition on each outgoing transition which checks the invariant, and also adding another transition which goes to a bad state upon the violation of the invariant.

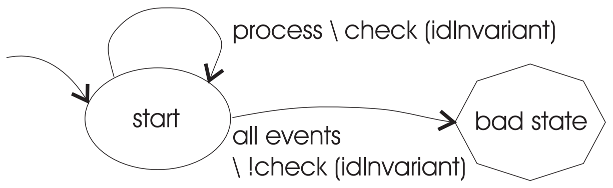


Figure 6.5: The implication of invariants on the automata.

To illustrate this, we will use an example. Consider the following snippet of LARVA script with an invariant and a transitions:

```
FOREACH (Transaction t)
{
  INVARIANTS {
    Long idInvariant = t.getID();
  }
  ...
  TRANSITIONS {
    start -> start [ process ] [ enable idInvariant ]
  }
}
```

The invariant checks that the return value of the method *getID* never changes throughout all the life cycle of the transaction. The invariant is named *idInvariant* and is enabled by the construct *enable* upon a particular transition. As soon as the invariant is enabled, the automaton modifies itself — the automaton with originally one transition and no conditions becomes as shown in Figure 6.5.

More concretely, an invariant is monitored using two variables: a boolean and an object which stores the last known value of the monitored attribute. The boolean variable is initially false, but it is set to true as soon as the invariant is enabled. The enabling of an invariant is always associated with a transition. Upon enabling the invariant, the current value of the invariant is stored in the designated variable. The following code is performed upon taking the transition which enables the invariant in the above example:

```
else if ((_occurredEvent(_event,1/*setID*/)))
{
  _state_id_settingAmount = 3; //moving to state ...
  idInvariant_enb = true;
  idInvariant_temp = t.getAmount();
  ...
}
```

Once the invariant is enabled and a value of the method is available, this is compared each time the automaton is retrieved from the hash table of active automata (before taking a transition). The following code shows the comparison taking place between the current value of *getID* and the value stored in the designated temporary variable:

```
_cls_property tmp = _cls_property_instances.get(_inst);
if ( tmp.idInvariant_enb && !tmp.idInvariant_temp.equals(t.getID()))
{
  _cls_property.pw.println(" Invariant Check: idInvariant Failed:
    t.getID()!: " + new _BadStateException().toString());
}
```

6.7 Real-Time in Larva

In this section we will explain how Larva treats real-time in its current implementation, and discuss the challenges of runtime verification of real-time properties.

Larva provides a clock construct which can trigger events after at particular time intervals. The clock can also be used to measure the time elapsed between occurrences of events. This is done by resetting the clock at the occurrence of the first event and then read the clock value when the second event occurs. Larva clocks are implemented as a Java thread which uses the *wait* operation to allow the required time to pass without keeping hold of the CPU. This approach is not totally accurate because of the scheduling mechanism used among Java threads. This does not guarantee the time at which the thread is given back control of the CPU. A solution to this problem is the use of the Java real-time virtual machine. However, this was not an option in our case because we cannot impose the real-time virtual machine on the monitored system.

6.7.1 Firing Events

A clock has a number of known points in time at which it is expected to trigger an event. This information can be extracted from the list of events concerning a particular clock. For example consider the following set of events on clock *clk*:

```
clkAT1() = {clk @ 1}
clkAT3() = {clk @ 3}
clkAT7() = {clk @ 7}
```

In this case, the clock *clk* is expected to trigger three events: one after a second, another after three seconds, and final event after seven seconds. Note that these duration are measured since the last clock reset. Each time the clock is reset, it starts measuring the time from zero and triggers the events accordingly. For efficient implementation, each of these time values (1, 3, 7) is put in an ordered list so that after each event the clock is immediately put in waiting mode for the next event. A clock is initialised by storing the current system time in a variable called *starting*. This is used as a reference point when the system time is later used to calculate the elapsed time. A loop is used to go through the ordered list of time values. At each iteration, the clocks waits for the necessary time for the next event. Taking the previous example, the clock should first wait for one second, then for two seconds (3-1) and finally for four seconds (7-3). Note that these operations are performed in a *synchronized* block to avoid data race conditions. The following code shows the code in the loop responsible for putting the clock to wait. *Registered* is the list of time values (in milliseconds) at which the clock is expected to trigger.

```
long nextMilli = registered.get(i);
synchronized (o) {
  long cur = System.currentTimeMillis();
  long tmp = nextMilli - (cur - starting);
  if (enabled && tmp > 0)
    o.wait(tmp);
  if (enabled && starting + tmp <= System.currentTimeMillis())
    new ClockEvent(this,nextMilli);
}
```

6.7.2 Resetting the Clock

The clock is reset by performing two simple steps: first the starting time of the clock is set to the current system time, and second, the loop which goes through the events is restarted. The only difficulty in doing these steps is to avoid data race conditions. For this purpose, when a reset is required, a boolean variable is set and the method *notify* is called. If the clock is currently waiting, the notification will cause the clock to exit the waiting mode and proceed with the loop. Note that the clock does not trigger the event unless the expected time has indeed elapsed. This is ensured by comparing the current time with the expected time of the event: `starting + tmp <= System.currentTimeMillis()`. The method *reset* is given below:

```
public void reset()
{
   resetc = true;
   synchronized (o) {
      o.notify();
   }
}
```

This above *reset* method causes the clock to exit the waiting state, but is not sufficient for the reset process. Thus, the code below sets the loop counter i to -1 (restarting the loop) and setting the clock starting time to the current system time.

```
if (resetc)
{
  i = -1;
  starting = System.currentTimeMillis();
  resetc = false;
}
```

6.7.3 Pausing and Resuming

Pausing and resuming work in a similar way to resetting. The method *pause* sets a boolean variable *paused* to true, while the method *resume* sets the same variable to false. In this case, the *notify* method is also used to free the clock from the *wait* method. When the loop of the clock finds the *paused* variable set to true, it enters a loop waiting to be released (when the *paused* variable is set to false). The time of entering this loop is very important because this has to be used to update the elapsed time of the clock — no time passes for the clock while it was paused. For this purpose the starting time of the clock is modified to reflect the time the clock was paused. The code is as follows:

```
whenPaused = System.currentTimeMillis();
synchronized (o) {
  while (enabled && paused)
  o.wait();
}
starting += System.currentTimeMillis() - whenPaused;
```

6.7.4 Time Comparison

Sometimes, conditions on transitions compare the current timer value (the time elapsed since the last reset) to some constant. In the current implementation, this is simply done by obtaining the current system time, deducting the timer's starting time, and comparing the result to the constant. At a first glance this works fine, but in the case where more than one transition is taken upon the same event, the illusion should be given that all the comparisons on the transitions are performed at the same instance. Theoretically, transitions should take no time to execute. Also, if for example, a transition is triggered at time 50 by a timer event, any comparison on that transition should be done as if the clock was temporarily paused at 50. However, the current implementation does not cater for these details due to lack of time. The necessary modifications would need to be done in future work. Currently, the code is as follows:

```
public long current()
{
  if (paused) return (whenPaused - starting);
  else return (System.currentTimeMillis() - starting);
}

public int compareToMillis(long milli)
{
  return new Long(current()).compareTo(milli);
}
```

6.8 Channels

Channels in LARVA are implemented as threads with a queue. The reason for using a queue, is that no events can be lost. Recall that since LARVA has timer events, any arbitrary number of channel events can be triggered at any moment. The channel waits until a notification is received, which signifies that an event has been placed on the queue. A timeout on the *wait* is also used to avoid any deadlocks in case the notification does not arrive. Upon receiving an event, this is broadcast to all the automata. The code which receives and processes the channel events is as follows:

```
while (queue.isEmpty() && enabled)
{
  synchronized (queue) {
    queue.wait(1000);
  }
  if (on)
    new ChannelEvent(this,queue.remove(0));
}
```

The code on the sending side simply places the event on the queue and sends a notification:

```
synchronized (queue) {
  queue.add(s);
  queue.notify();
}
```

6.9 Conclusion

An automated way of injecting code at particular points in a program executions, allows us to capture events which can be used to trigger transitions on automata. These automata are easily implemented as a class which can also keep a reference to the object (or objects) which form the context of the automaton. The actual logic of the automaton is implemented as a method with a series of if-then-else statements carrying out the required actions where appropriate. In the case of clocks, the events driving the automaton are not fired by the monitored system. Instead, a separate clock thread is used to trigger events upon particular timeouts. Similarly, channels are used so that automata can send events to each other. Upon each event, particular checks can be carried out on the context of an automaton to check that particular attributes of the monitored object remain unchanged.

This chapter has explained the low level details of LARVA. The following chapter will relate LARVA to other logics, formulating translations from these logics to LARVA. Translations preserve the advantages of the translated logics while the LARVA architecture can still be used.

Part III

Theory

7. Larva and Other Logics

7.1 Introduction

Duration calculus is sometimes more appropriate than DATEs to express certain properties. For example, duration calculus makes it very easy to express properties which must hold on all subintervals of an interval. This may not be so straightforward to do in DATEs. Thus, if a translation from duration calculus to DATEs is available, certain properties can be written in duration calculus and at the same time monitored by LARVA. However, the full duration calculus cannot, in general, be monitored. This leads us to translate a subset of duration calculus called counterexample traces and prove the correctness of the translation. Similarly, the Lustre language is also translated into DATEs to gain guarantees on the memory and temporal overheads induced by the monitors. Another subset of duration calculus — QDDC — is both useful to express particular properties and translatable into Lustre. Thus, QDDC also inherits the advantage of guaranteed overheads upperbound.

In this chapter, we will show the relationship of DATEs to two subsets of duration calculus: counterexample traces and QDDC. The translation of the former into DATEs is explained in Section 7.2, while Section 7.3 explains how the deterministic fragment QDDC can be monitored by LARVA by going through Lustre. The contract language is shown to be monitorable by LARVA in Section 7.4 while Section 7.5 concludes the chapter.

7.2 Larva, Duration Calculus and Phase Event Automata

In the chapter about real-time logics (Chapter 3), we have already given an explanation of the translation of a subset of duration calculus into phase event automata (originally given by Hoenicke [58]). In the next sections, we will give a short summary of the translation from duration calculus to phase event automata followed by the complete translation from phase event automata to DATEs.

7.2.1 Translating Duration Calculus to Phase Event Automata

The translation of a counterexample formula starts by representing the formula into mathematical structures. The negation at the start of the counterexample formula is ignored. The generated phase event automaton will, in fact, accept the prohibited formula. The equivalent mathematical structure of a formula is a *trace* while the equivalent of a phase is a *phase*. Thus, a *trace* is a sequence of *phase* structures, where each *phase* corresponds to a phase in the formula. The *phase* structure also includes the *entry events* of the corresponding phase in the formula. These include any events which should occur before a phase can be entered.

Each phase is numbered according to its position in the sequence. A phase is a lowerbound ($phase \in LB$) if and only if it has a lowerbound time constraint of the form $\ell > t$ or $\ell \geq t$. Similarly, a phase is an upperbound ($phase \in UB$) if and only if it has an upperbound time constraint of the form $\ell < t$ or $\ell \leq t$.

During the execution of an automaton, a phase can belong to a number of the following sets: *in*, *wait*, *gteq*, and *less*. A phase is in *in* ($phase \in in$) if it is currently *active*, i.e. the phases up to this phase have been detected. A phase is in *wait* ($phase \in wait$) if it is an element of LB and it is active, and the duration of the phase is less than its lowerbound. A phase is in *gteq* ($phase \in gteq$) if the bound is of the form $\ell \geq t$ and the phase is in *wait*. A phase is in *less* ($phase \in less$) if the bound is of the form $\ell < t$ and the phase is active, and the duration is less than the bound.

A *location* in the generated phase event automaton is labelled by the phases which are an element of *in*, *wait*, *gteq*, and *less* accordingly. If a location contains the last phase of the trace in its *in* and this phase is not in *wait*, then the last phase has been detected. This means that a counterexample has been detected, i.e. the original counterexample formula has been violated.

For example, consider the formula $\neg\ (\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil)$. The equivalent phase event automaton is shown in Figure 7.1.

7.2.2 Translating Phase Event Automata into DATEs

Assumptions

There are a number of assumptions without which the translation to DATEs will not hold:

1. It is assumed that the last phase is a true-phase. Without this assumption the bad states translation would not work. The reason is that for a non-true-phase a number of conditions must be checked to ensured the phase is completed. For a true-phase it is enough that the phase number is an element of the label of the location.

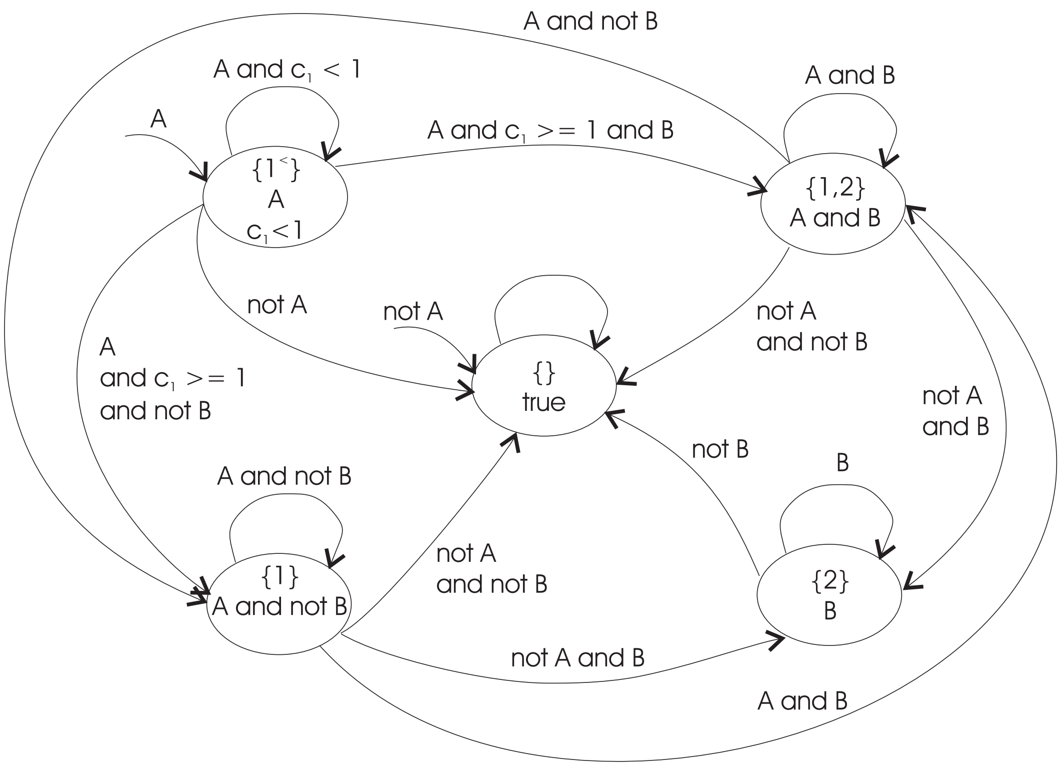


Figure 7.1: The phase event automaton equivalent to the formula $\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil$.

2. We also assume totality of the automaton so that there exists a transition for any possible state variable configuration. In other words, for each location of the automaton, the disjunction of the conditions on the outgoing transitions should be equivalent to *true*.

Translation

In this section, we will give the translation of a phase event automaton into a DATE. First, we will list a number of functions required for the translation. Then, we give the construction of a DATE from a phase event automaton. Finally, we give the translation of a phase event automaton configuration into a DATE configuration.

Preliminary Functions First we define the following two functions which will be used in the translation. These functions take elements from the phase event automaton and return elements for the DATE. To make this clear we will denote the elements from the phase event automaton with $^{\mathcal{P}}$ and elements from the DATE with $^{\mathcal{D}}$. Furthermore, we refer to the nodes of the phase event automaton as *locations* and those of the DATE as *states*.

Φ generates exactly one DATE event for each variable and event in $NAME^{\mathcal{P}}$.

The DATE event will trigger upon the rising edges of the variable $NAME^{\mathcal{P}}$ or upon the occurrence of an event $NAME^{\mathcal{P}}$. For the translation, we will require the union of all the generated events. Thus, for convenience, this collection will be referred to as $AllEvents^{\mathcal{D}}$.

Γ generates exactly one DATE event for each clock constraint in the phase event automaton. The event will be based on the constant to which the clock is compared. For example, for a constraint: $c_1 \leq 10$, an event will be raised by clock c_1 ten time units after its last reset. This will be useful to detect the time at which the constraint $c_1 \leq 10$ will no longer be satisfied.

Another function required for the translation is the function RR which given a clock invariant CI, and a set X of clocks, returns the *strict*[1] clock invariant CI'. Furthermore, the clock constraints which apply to any of the clocks in the set X are modified as follows: the current value of the clock is added to the bound. The purpose is that the modified constraint will behave as if the clock has been reset. This is very useful because, in the case of DATE, the condition has to be checked before the clocks are reset while in a phase event automaton, it is the other way round.

Finally, we require two other functions. DATEs do not allow events in the conditions on the transitions while phase event automata do. Thus, we need these functions to bridge the difference. This is achieved in two steps. In the first step, for each event, we create a boolean variable which is true if the event has occurred and false if it has not. In other words, it is true if the active configuration contains the particular event and false if it does not. In the second step, we modify the transitions' conditions by replacing each event with its corresponding boolean variable. The first function will be referred to as BV while the second will called RE. BV will take a set of events and returns a set of variables while RE while take a condition with events and returns a condition without events.

Constructing a DATE The translation (Ψ) from a phase event automaton $\mathcal{P} = \langle P,\ V,\ A,\ C,\ E,\ s,\ I,\ E_0 \rangle$ to a DATE $\mathcal{D} = \langle Q,\ q_0,\ \rightarrow,\ B,\ A \rangle$ is defined as follows:

[1]Note that the *strict* is a function which replaces constraints of the type $\leq$ and $\geq$ into $<$ and $>$ respectively, while keeping the rest of the constraints unchanged.

$$\begin{array}{rcl}
Q & = & P \cup \{s_0\} \\
q_0 & = & s_0 \\
\rightarrow & = & \mathit{for\ each}\ (p, g, X, p') \in E^{\mathcal{P}},\ \mathit{we\ construct}: \\
 & & (p, \mathit{AllEvents} \cup \Gamma(I(p)), \\
 & & (s(p') \wedge RR(I(p'), X) \wedge RE(g)), (X, \epsilon, \epsilon), \epsilon, \epsilon, p'). \\
 & & \mathit{The\ following\ transitions\ handle\ initialization\ locations}: \\
 & & \mathit{for\ each}\ (g, p) \in E_0\ \mathit{we\ construct}:\ (s_0, \mathit{init}^2, g, \epsilon, \epsilon, \epsilon, p). \\
B & = & \{p \in P \mid \#(tr) \in p.in\}
\end{array}$$

Given a phase event automaton $\mathcal{P}$, we create an equivalent state for each location in set P. We create a new state s_0 which will be the starting state of the DATE. For each edge of the phase event automaton, we create a corresponding transition by obtaining the relevant events using the functions Φ and Γ, the condition as the conjunction of the guard and the invariant of the destination location, and the resets remain unchanged. The last group of transitions are created to correspond with the initial locations of the phase event automaton. The set of bad states includes all the location of the phase event automaton which have detected the whole trace. Recall that we are monitoring the violation (not the satisfaction) of the formula. The construction of transitions is depicted in Figure 7.2.

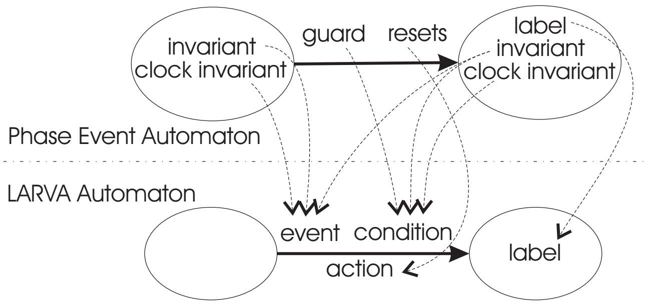


Figure 7.2: Generating DATE transitions from a phase event automaton.

The above translation is labelled Ψ such that applying the translation on a phase event automaton $\mathcal{P}$ yields a DATE $\mathcal{D}$: $\mathcal{D} = \Psi(\mathcal{P})$.

Translating Configurations A *configuration* of a phase event automaton is a quintuple (p, Y, β, γ, t) where p is the location, Y is a set of events, β is a

[2]This refers to the *init* event which triggers automatically at the start of the monitoring system (see Subsection 4.3.1)

valuation of variables, γ is a valuation of clocks and t is the duration spent in location p.

A *configuration* of a DATE is (X, θ, q) where X is a set of events, θ is the system state, and q is the automaton state. Such a configuration moves to the next configuration according to a timer configuration T and timer update t'. This is written as follows: $(X, \theta, q) \Rightarrow^{T}_{t'} (X', \theta', q')$. Note that T and t' are written on the arrow but for the purpose of proofs these will be written in the brackets. Thus, the first configuration would become: (X, θ, q, T, t').

Overloading the translation Ψ to be applicable to configurations, a DATE configuration $cp^{\mathcal{D}}$ is obtained by applying Ψ on a phase event automaton configuration $cp^{\mathcal{P}}$: $cp^{\mathcal{D}} = \Psi(cp^{\mathcal{P}})$.

For any phase event automaton configuration (p, Y, β, γ, t), the corresponding DATE configuration is: $(Y \cup @\Phi \cup @\Gamma, \beta, p, \gamma, +t)$, where the symbol @ represents all the events occurring during the duration of that configuration for all the events being monitored, i.e. all the events generated by the functions Φ and Γ respectively. The timer increment t is converted to the function $+t$ because the timer updates of DATEs are functions.

The initial configuration of a DATE is written as $icp^{\mathcal{D}} = (\emptyset, \theta, q_0, \gamma, 0)$. In a translation, $q_0 = s_0$ and $\theta = \beta$ where β is the initial configuration of the phase event automaton.

Example

We use the example given earlier where the phase event automaton which monitors the formula $\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil$ was generated. This was then translated into a DATE. This result is shown in Figure 7.3.

7.2.3 Proof of Correctness of Translation

The proof of showing that the translation from a phase event automaton to a DATE is correct is divided into five steps:

1. First, we will show that a one-step run for a phase event automaton, has a corresponding one-step run in the corresponding DATE. Note that this one-step run does not start from an initial state and assumes that the step is a non-stuttering step.

2. Once we have proved the translation for one step, we prove the translation for any number of steps given that the run does not start from an initial state and does not include stuttering steps.

3. The next step in the proof is to show that a non-stuttering run starting from an initial state, has a corresponding run in the corresponding DATE, also starting from an initial state.

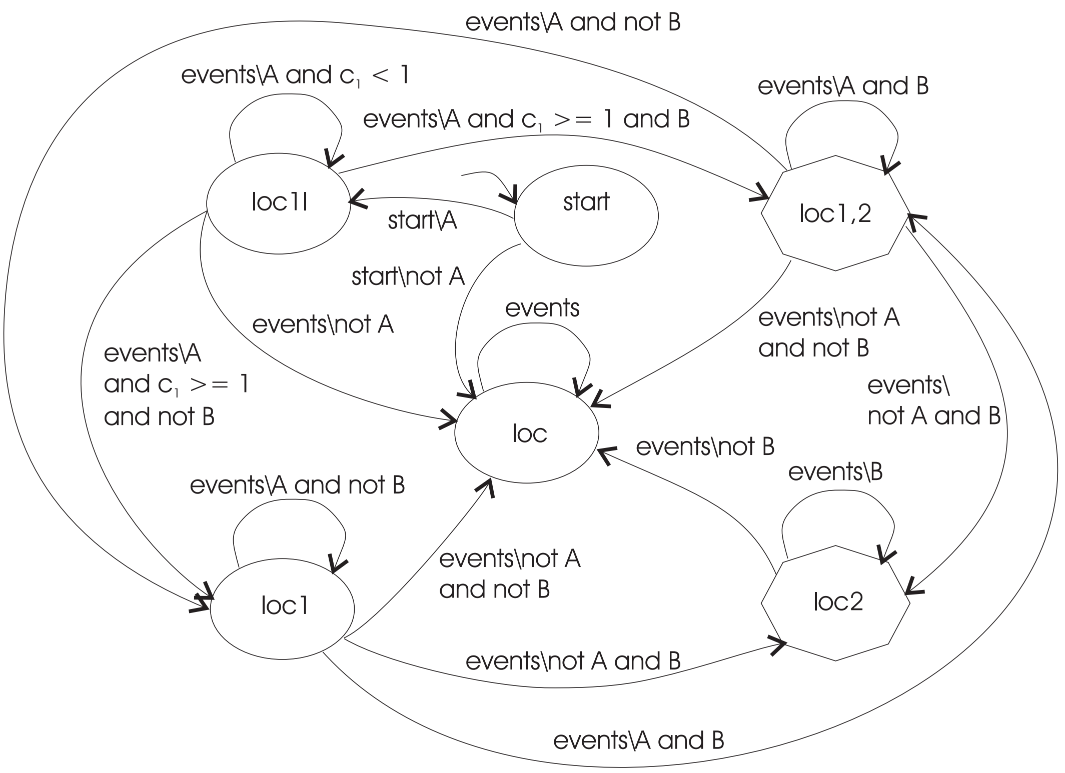


Figure 7.3: The DATE equivalent which monitors the property $\lceil A \rceil \wedge \ell > 1 \frown \lceil B \rceil$.

4. Up to this point, we have shown that for every non-stuttering run in a phase event automaton there is a corresponding run in the DATE generated by the translation. Thus, we also need to show that there are no other configurations which can be reached by the DATE, except the one obtained by the translation Ψ.

5. The final step is to lift the condition of considering only non-stuttering runs and prove the translation for all the runs of the automata.

The following steps correspond to the above five steps:

1. As the first step of the proof, we will show that each one-step run in a phase event automaton has a corresponding one-step run in the DATE. This is shown in the following proposition:

   Proposition 7.2.1. If a non-stuttering transition can be taken in a phase event automaton $\mathcal{P}$ from a non-initial configuration cp to another configuration cp', then there exists a corresponding one-step run in the DATE $\Psi(\mathcal{P})$ which starts from the corresponding configuration $\Psi(cp)$ and ends with configuration $\Psi(cp')$ (Note that in a non-stuttering run, the state of the automaton changes from a configuration to the next.):

$$\mathcal{P}/cp \overset{trans}{\Longrightarrow} \mathcal{P}/cp' \Rightarrow \Psi(\mathcal{P}/cp) \overset{\Psi(trans)}{\Longrightarrow} \Psi(\mathcal{P}/cp')$$

Proof. Let us assume that there exists a single-step run in a phase event automaton $\mathcal{P}$ which starts from configuration cp and ends at configuration cp':

$$\mathcal{P}/cp \overset{trans}{\Longrightarrow} \mathcal{P}/cp'$$

By definition of a configuration:

$$cp = (p, Y, \beta, \gamma, t), \text{ and } cp' = (p', Y', \beta', \gamma', t')$$

By translation Ψ of a configuration:

$$\Psi(cp) = (Y \cup @\Phi \cup @\Gamma, \beta, p, \gamma, +t)$$

and

$$\Psi(cp') = (Y' \cup @\Phi \cup @\Gamma, \beta', p', \gamma', +t')$$

By definition of a transition (assuming that cp is a non-initial configuration):

$$trans = (p, g, X, p')$$

By translation of a transition $\Psi(trans)$:

$$\Psi(trans) = (p, \Phi(s(p)) \cup \Gamma(I(p)) \cup \Phi(s(p')) \cup \Gamma(I(p')),$$
$$(s(p') \wedge RR(I(p'), X) \wedge g), (X, \epsilon, \epsilon), \epsilon, \epsilon, p')$$

For a DATE transition to be taken, the source location p must be the same location of the configuration. Thus, this condition is satisfied.

The second condition is that the event of the transition is triggered. By the assumption that the run is non-stuttering, one of the following is true for two subsequent configurations: (i) a global variable changed; (ii) an event occurred; or (iii) the invariant of the location is no longer satisfied. Considering the first two possibilities, we have included variables' changes and events in *AllEvents*. Thus, a variable change or an event, will cause one of the events in the collection *AllEvents* to trigger. For the third possibility, the invariant of the location is no longer satisfied either because of a variable change or because the clock has exceeded its bound. Variable changes have already been catered for. For the second case, the clock exceeds its bound when it reaches the bound. At this moment, an event generated by the function $\Gamma(I(p))$ triggers. Thus, we have considered all the possibilities and it is clear that an event will trigger:

$$Y \cup @\Phi \cup @\Gamma \vDash Y \cup \mathit{AllEvents} \cup \Gamma(I(p))$$

The third condition for a DATE transition to trigger is that the condition holds. For this part of the proof, we will use the three conditions which hold for each step in a phase event automaton run:

$$(\beta', \gamma + t, Y') \vDash g,\ \beta' \vDash s(p') \text{ and } (\gamma + t) \vDash \mathit{strict}(I(p'))$$

By the translation $\Psi(trans)$ we get the following condition on the DATE transition:

$$s(p') \wedge RR(I(p'), X) \wedge RE(g)$$

Taking the condition in parts: $s(p')$ holds because β' is kept in the DATE configuration by the translation Ψ. $RR(I(p'), X)$ holds because $(\gamma + t) \vDash \mathit{strict}(I(p'))$, and by definition, $RR(I(p'), X)$ is less restrictive than $\mathit{strict}(I(p'))$, thus $(\gamma + t) \vDash RR(I(p'), X)$. The function RR has the purpose of adjusting the constraints on clocks which will be reset on the transition action. This adjustment is required because, in DATEs, the clock constraint is applied before the clock is reset, while in the phase event automaton, the invariant is applied after the clock is reset. Finally, $RE(g)$ will be satisfied because $\beta', \gamma + t$, Y' satisfy g, β' and $\gamma + t$ were kept in the DATE configuration, and for each event in Y' we have created a corresponding variable (using BV) which is used by the function RE to produce $RE(g)$.

Thus, we conclude that the translated transition is taken and the corresponding clocks in set X are reset. Consequently, for each single-step run in the phase event automaton, there is a corresponding run in the generated DATE.

□

2. In this step, we will use the previous proof of one-step runs to prove that a non-stuttering run, with any number of steps in a phase event automaton, has a corresponding run with the same number of steps in the DATE. The assumption that both runs start from a non-initial configuration is still required.

   Theorem 7.2.1. Given a phase event automaton $\mathcal{P}$: starting from a configuration *cp*, another configuration *cp'* can be reached in a number of steps n: $cp \overset{\mathcal{P}}{\Longrightarrow}_n cp'$. The generated DATE $\Psi(\mathcal{P})$ can start from the corresponding configuration $\Psi(cp)$ and reach $\Psi(cp')$ in a number of steps.

$$cp \overset{\mathcal{P}}{\Longrightarrow}_n cp' \Rightarrow \Psi(cp) \overset{\Psi(\mathcal{P})}{\Longrightarrow}_n \Psi(cp')$$

   Proof. Using induction on n:

   Prove theorem for $n = 0$:

   Note that in zero number of steps ($\overset{\mathcal{P}}{\Longrightarrow}_0$) the configuration does not change.

$$\begin{array}{rl} & cp \overset{\mathcal{P}}{\Longrightarrow}_0 cp' \\ \Longrightarrow & \text{By definition of } \overset{\mathcal{P}}{\Longrightarrow}_0 \\ & cp \overset{\mathcal{P}}{\Longrightarrow}_0 cp \\ \Longrightarrow & \text{By definition of } \overset{\Psi(\mathcal{P})}{\Longrightarrow}_0 \\ & \Psi(cp) \overset{\Psi(\mathcal{P})}{\Longrightarrow}_0 \Psi(cp) \end{array}$$

Inductive hypothesis: assume theorem holds for $n = k$.

Prove theorem for $n = k + 1$:

$$\begin{array}{rl} & cp \overset{\mathcal{P}}{\Longrightarrow}_{k+1} cp' \\ \Longrightarrow & \text{By definition of } \overset{\mathcal{P}}{\Longrightarrow} \\ & cp \overset{\mathcal{P}}{\Longrightarrow}_k cp'' \textit{ and } cp'' \overset{\mathcal{P}}{\Longrightarrow}_1 cp' \\ \Longrightarrow & \text{By inductive hypothesis} \\ & cp \overset{\Psi(\mathcal{P})}{\Longrightarrow}_k cp'' \\ \Longrightarrow & \text{There is a one-step run in } \Psi(\mathcal{P}) \text{ for each one-step run in } \mathcal{P} \\ & \text{(Proposition 7.2.1)} \\ & cp'' \overset{\Psi(\mathcal{P})}{\Longrightarrow}_1 cp' \\ \Longrightarrow & \text{Combining together we get} \\ & cp \overset{\Psi(\mathcal{P})}{\Longrightarrow}_k cp'' \textit{ and } cp'' \overset{\Psi(\mathcal{P})}{\Longrightarrow}_1 cp' \\ \Longrightarrow & \text{By definition of } \overset{\Psi(\mathcal{P})}{\Longrightarrow} \\ & cp \overset{\Psi(\mathcal{P})}{\Longrightarrow}_{k+1} cp' \end{array}$$

□

3. In this part of the proof, we will also include runs which start from an initial state. Note that the run must still be a non-stuttering run. The idea of the proof is to show that the first transition taken in a phase event automaton has a corresponding behaviour in the DATE. Subsequently, we connect the first step of a run to the rest of it and we will extend the result of the previous theorem to apply to complete runs.

Theorem 7.2.2. Given a phase event automaton $\mathcal{P}$, starting from an initial configuration *icp*, another configuration *cp* can be reached in a number of steps n. The generated DATE $\Psi(\mathcal{P})$ can start from the initial DATE configuration $\Psi(icp)$ and reach $\Psi(cp)$ in a number of steps *n+1.*

$$\begin{aligned}
& (icp \overset{\mathcal{P}}{\Longrightarrow}_n cp) \Rightarrow (\Psi(icp) \overset{\Psi(\mathcal{P})}{\Longrightarrow}_{n+1} \Psi(cp)) \\
\Longrightarrow\ & \text{By definition of a phase event automaton} \\
& icp = (p, \emptyset, \beta, \gamma, t),\ \ s.t.\ \beta \models g \ where\ (g, p) \in E_0 \\
\Longrightarrow\ & \text{By definition of a DATE initial configuration} \\
& icp^{\mathcal{D}} = (\emptyset, \beta, s_0) \\
\Longrightarrow\ & \text{By translation } \Psi \text{ of initial transitions, } \textit{init} \text{ event, and } \beta \models g \\
& icp^{\mathcal{D}} \overset{\Psi(\mathcal{P})}{\Longrightarrow}_1 \Psi(icp) \\
\Longrightarrow\ & \text{By Theorem 7.2.1} \\
& (icp \overset{\mathcal{P}}{\Longrightarrow}_n cp) \Rightarrow (\Psi(icp) \overset{\Psi(\mathcal{P})}{\Longrightarrow}_n \Psi(cp)) \\
\Longrightarrow\ & \text{By catenation of } \overset{\Psi(\mathcal{P})}{\Longrightarrow} \\
& icp^{\mathcal{D}} \overset{\Psi(\mathcal{P})}{\Longrightarrow}_1 \Psi(icp) \overset{\Psi(\mathcal{P})}{\Longrightarrow}_n \Psi(cp) \\
\Longrightarrow\ & \text{By definition of } \overset{\Psi(\mathcal{P})}{\Longrightarrow} \\
& icp^{\mathcal{D}} \overset{\Psi(\mathcal{P})}{\Longrightarrow}_{n+1} \Psi(cp)
\end{aligned}$$

4. In the previous steps, we have shown that for every non-stuttering run in a phase event automaton, there is a corresponding run in the translation-generated DATE. To prove the translation correct, we also need to show that there is no other configuration which the DATE can reach except the one generated by the translation. More formally, we have shown completeness but not soundness.

   Proposition 7.2.2. The DATE generated by the translation Ψ has only one run for each corresponding non-stuttering run in the phase event automaton.

   Proof. This is easy to show because DATEs are deterministic by their definition (see Section 4.3). Thus, there is no possibility of reaching any configuration other than that obtained by the translation (proved in Theorem 7.2.2).

   □

5. The final step is to lift the condition that only non-stuttering runs can be translated.

Proposition 7.2.3. Every run in the phase event automaton has a single corresponding run in the generated DATE.

Proof. The solution is to use the fact that phase event automata are stuttering invariant (see Hoenicke's [58] Lemma 4.10 on pg. 78). This means that for every stuttering run, we can construct a non-stuttering run by removing the stuttering configurations. All non-stuttering runs have been shown in Proposition 7.2.2 to have a single corresponding run in the corresponding DATE. Thus, this final step concludes our proof of the correctness of the translation. □

7.2.4 Future Work

In this section, we will give an account of possible developments in the work presented above.

Context

With very little modification, the dynamic aspect of DATEs can also be exploited by translated counterexample traces. In other words, the counterexample trace will be applied on each individual object rather than on a global context. This can be achieved by introducing constructs similar to the *FOREACH* construct in Larva. Otherwise, the modification can be done manually on the Larva script output of the translation from a counterexample trace.

Bounded Memory

In the future, the memory upperbound for phase event automata monitoring counterexample traces can be investigated. Our intuition is that the required memory for a phase event automaton does not increase during the monitoring. Thus, the upperbound can be known at compile time. The reason is that the phase event automaton uses a fixed number of clocks, a fixed number of state variables, and a fixed number of states.

Proposed Extension — Experiments with Counters

We have done a number of experiments to add counters to phase event automata. The idea is to introduce counters as a kind of clocks which count occurrences of events rather than the real time. Therefore, each phase is allowed to have a restriction on one of these counters. Two problems arise here: the first is that this inherently creates a number of additional states which the current algorithm cannot handle (i.e. one timer can be satisfied, the other may not — there are 4 possibilities); the second is that there are cases where the number of timers required would be infinite. Consider the formula: $\mathit{true} \frown \mathit{len} < 10\ \mathit{and}\ \mathit{cnt}(\mathit{badlogin}) > 3$ — there are an unbounded number of intervals whose length is less than 10 for which the counter can exceed 3. Although we have tried some experiments with counters at this stage, it is too early to give any results. This may be an interesting addition to phase event automata for the future.

7.3 DATEs, QDDC and Lustre

7.3.1 Synchronous Programming

When employing runtime verification, we usually do so for systems where errors are highly undesirable. The simple applications which we use daily are not usually verified but software controlling railways or other expensive machinery are. Most systems which we would like to verify fall under the class of reactive systems. Such systems react to inputs from their environment by changing some outputs at real-time. A class of languages which have been specifically designed for programming reactive systems are known as synchronous languages. The synchronous nature of these languages lies in the fact that the reaction time of the system is considered to be negligible (i.e. the system reacts instantaneously to the inputs). One such language is Lustre [46]. A very important advantage of Lustre is that the memory and temporal requirements for the monitoring code can be measured at compile time. This is very important in security-critical systems since it enables us to give guarantees on the upper bound of the monitoring overhead. A small example of Lustre code is shown below:

```
node BadAccess(w,r,i,o:bool)returns(bw,br:bool);
var l:bool;
let
  l = if (o) then false
    else if (i) then true
    else false->pre(l);
  bw = w and not(l);
  br = r and not(l);
tel
```

What this piece of code does is that it monitors four events: write (w), read (r), login (i) and logout (o). It keeps track of whether a user is logged in or logged out in the local variable l. If a read or write event occurs while login (l) is false, br (or bw respectively) is set to true.

Another interesting point of view is that Lustre can be considered as an executable temporal logic [47]. This has motivated the specification of both the system and its properties to be in Lustre [47]. A similar concept which is introduced is the concept of *observers* [48]. An observer is a program which checks that the main program keeps to its specification. Hence, the verification problem then simply involves: (i) the composition of the two programs, and (ii) ensuring that the observer never reaches an undesired state. This concept has been extended some years later [79], adding the expressive power of regular expressions.

More specifically, properties including the regular constructs *sequence* and *iteration* are translated into an equivalent boolean dataflow network in the language Lustre. Another recent development was the synthesizing of Lustre observers from QDDC [45]. Recall that QDDC is a very expressive interval logic based on the discrete model of time. Furthermore, a deterministic subset of QDDC was used to show how this can be verified using deterministic observers. Imagine we want to implement the $\sum q$ QDDC operator which counts the number of times q was true during a particular interval. Using the simple Lustre code shown below, we can count the number of times q was true since the start of a particular period (indicated by p being true).

```
after_p = p or (false -> pre(after_p));

nb_q_since_p = if p then (if q then 1 else 0)
               else if after_p then
     (pre(nb_q_since_p))+(if q then 1 else 0)
               else 0
```

7.3.2 Translating QDDC to Lustre

The conversion from QDDC into Lustre is already given by Gonnord et al. [45]. Here we will give a short description of the process. For each QDDC formula we need to specify a node which returns true if the formula is satisfied and false if otherwise. This is achieved by using a number of basic Lustre nodes. These are then called by more complex nodes. The following list includes the basic nodes required for the conversion:

Node name	Description
$after(p)$	Returns true at the occurrence of p and afterwards
$strict\_after(p)$	Returns true after p but not at the occurrence of p
$first(p, b)$	Returns true at the first occurrence of p *after* b occurs
$always\_since(p, b)$	Returns true if b has not been true, or p has always been true *after* b
$age(p, b)$	Counts the number of instances p has been continuously true *after* b
$nb\_since(p, b)$	Counts the number of occurrences of p *after* b

Using these basic nodes we construct more complex ones. For example, consider the QDDC formula $begin(p)$ which is satisfied if the first state of the interval satisfies p. The acceptor of $begin(p)$ if at the beginning of the interval (when b is true) p is also true. Thereafter, the acceptor should return true till the end of the interval. For this purpose, we use the node *after* on the conjunction of p and b. Thus, the acceptor for $begin(p)$ will be $after(b \textit{ and } p)$. However, note that the input of *begin* can be a boolean expression. To cater for this possibility, the acceptor of *begin*(P) may require a number of inputs. These inputs will be represented by $\mathcal{I}$. The resulting acceptor, denoted by $\mathcal{A}_{begin(P)}(b, \mathcal{I})$, will be defined by *after*$(b$ *and* $\mathcal{A}_P(\mathcal{I}))$, where $\mathcal{A}_P$ represent the acceptor of P.

The above example of creating the *begin* acceptor shows how acceptors are used by other acceptors. Adopting this approach we can recursively define acceptors for all the propositions. The simplest acceptor is the one which accepts an atomic proposition p — if p is true, then p is accepted; otherwise p is not accepted. The complete list is defined as follows:

Proposition	Acceptor
p	p
$\neg P$	$not\ \mathcal{A}_P(\mathcal{I})$
$P_1 \wedge P_2$	$\mathcal{A}_{P_1}(\mathcal{I})\ and\ \mathcal{A}_{P_2}(\mathcal{I})$
$P_1 \vee P_2$	$\mathcal{A}_{P_1}(\mathcal{I})\ or\ \mathcal{A}_{P_2}(\mathcal{I})$

Next, we will consider acceptors for the fragment G (see definition of the deterministic QDDC fragment Section 3.4):

G formula	Acceptor
$begin(P)$	$after(b\ and\ \mathcal{A}_P(\mathcal{I}))$
$\lceil\lceil P \rceil$	$strict\_after(b)\ and\ pre(always\_since(\mathcal{A}_P(\mathcal{I}), b)$
$\eta \leq c$	$nb\_since(true, b) \leq c$
$\Sigma P \leq c$	$nb\_since(\mathcal{A}_P(\mathcal{I}), b) \leq c$
$age(P) \leq c$	$age(\mathcal{A}_P(\mathcal{I}), b) \leq c$
$G_1 \wedge G_2$	$\mathcal{A}_{G_1}(\mathcal{I})\ and\ \mathcal{A}_{G_2}(\mathcal{I})$
$G_1 \vee G_2$	$\mathcal{A}_{G_1}(\mathcal{I})\ or\ \mathcal{A}_{G_2}(\mathcal{I})$

Finally, we define the fragment F:

F formula	Acceptor
$end(P)$	$after(b)\ and\ \mathcal{A}_P(\mathcal{I})$
$G\ then\ F$	$\mathcal{A}_F(first(not\ \mathcal{A}_G(b, \mathcal{I}), \mathcal{I}))$
$F_1 \wedge F_2$	$\mathcal{A}_{F_1}(\mathcal{I})\ and\ \mathcal{A}_{F_2}(\mathcal{I})$
$\neg F$	$not\ \mathcal{A}_F(\mathcal{I})$

7.3.3 Translating Lustre into DATEs

A Lustre program is a symbolic automaton. Therefore, the translation to DATEs is very straight forward. The whole translation is as follows:

- The Lustre program is flattened in one node.

- An equivalent state *main* is created in the DATE with a single circular transition whose action is the code in the Lustre flattened node.
- To handle the initialization of the Lustre variables, a starting state is created in the DATE with a single outgoing transition to state *main*. The action on this transition will include all the necessary initialisations.
- The only remaining part is to set up the events in the LARVA script. First, we need an event which represents the initialisation of the system so that the initialisation transition is taken. Secondly, we need to relate the input streams in the Lustre program to events in LARVA. Upon any of these events, input streams are set accordingly, and the transition of the *main* state should be taken.

The end result of the translation process is shown in Figure 7.4 and illustrated with an example in the next section.

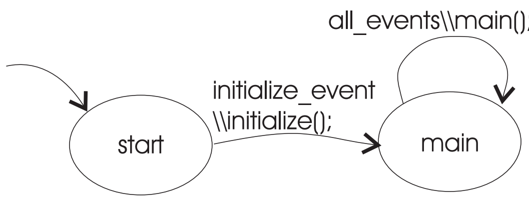


Figure 7.4: The DATE equivalent to a Lustre program.

7.3.4 Example

Consider the mine pump example. The following property written in the QDDC fragment states that 1000 time units after the water reaches the dangerous level, the alarm should have started. When the alarm stops, the water should no longer be over the dangerous level. It is written as:

$$\begin{array}{l} age(D) < 1000 \ \mathit{then} \\ ((\lceil\lceil Alarm\rceil \vee (begin(Alarm) \wedge (len \leq 0))) \ \mathit{then} \\ (\lceil\lceil \neg\, D\rceil \vee (begin(\neg\, D) \wedge (len \leq 0)))) \end{array}$$

When converted into Lustre we obtain the code below:

```
then_19(_b:bool;_rt_clock:int;D:bool;Alarm:bool)returns(_p:bool);
let
_p = then_16_34;
...
strict_after_88 = if (false->pre_1_89) then (true) else (false->pre_2_90);
pre_4_27 = temptime_26;
```

```
...
pre_2_90 = strict_after_88;
tel
```

This in turn generated the LARVA code shown below:

```
GLOBAL {

  VARIABLES {
    Clock _clock;
    boolean pre_8_30;
    ...
    boolean never_p_31;
  }

  EVENTS {
    _b_event(boolean _b, boolean D, boolean Alarm) = {***.***()}
      where { _b = true; D = false; Alarm = false;}
    D_event(boolean _b, boolean D, boolean Alarm) = {***.***()}
      where { _b = false; D = true; Alarm = false;}
    Alarm_event(boolean _b, boolean D, boolean Alarm) = {***.***()}
      where { _b = false; D = false; Alarm = true;}
    initializationEvent() = {***.***()}
    periodicEvent(boolean _b, boolean D, boolean Alarm)
      = {_b_event | D_event | Alarm_event}
  }

  PROPERTY then_8 {

    STATES {
      NORMAL { lustre }
      STARTING { initilization }
    }

    TRANSITIONS {
      initialization -> lustre
        [initializationEvent\ \
         pre_4_27 = 0;
         ...
         pre_0_55 = false;]

      lustre -> lustre
        [periodicEvent\ \
         _rt_clock = _clock.current_long();
         p_0_51 = D;
         never_p_31 = (_b)?(true):((pre_8_30)?(false):(pre_9_29));
         ...
         pre_0_55 = after_54;]
    }
  }
}
```

7.3.5 Modifying the Framework to Handle Real-Time

To modify the framework to handle real-time, no extensions are required to Lustre. The only requirement is a stream of timestamps. At each occurrence of an event, we have a timestamp available which gives the current time as an integer in a given time unit (e.g. milliseconds). We will call this stream *rt_clock* and it will be available to all the nodes as if it was another parameter.

To incorporate this with the work done by Gonnord et al. [45], we give the definitions of the real-time integral and the real-time *age*. The real-time integral given in the code below returns the total number of time units for which the boolean expression P was true.

```
node integral (rt_clock:int; P: bool; b: bool) returns (realtime: int);
let
  realtime = if (strict_after(b) and false -> pre P)
    then (0 -> pre(realtime)) + rt_clock - (0 -> pre(rt_clock))
    else (0 -> pre(realtime));
tel
```

The Definition of *age*

The definition of the real-time *age* is quite tricky. The problem is to decide when the *timer* of *age* is started. If we reset it to zero at the first occurrence, then the behaviour would be different from the non real-time *age*. Consider for example $age(P) > 0$. This would evaluate to true in the first occurrence of P using the non real-time version of *age*. However, using the real-time version, $age(P) > 0$ evaluates to false at the first occurrence of P. The alternative would be to start the timer from the last time P was false, but this does not seem very reasonable. Thus, the timer is reset at the first occurrence with the restriction that $age(P) > 0$ is not used to check whether P is true at the current point in time. Therefore, we propose the node below:

```
node age (rt_clock:int; P: bool; b: bool) returns (realtime: int);
var
  temptime: int;
let
  temptime = if (not (after(b) and (p))) then rt_clock else (0->pre(temptime));
  realtime = rt_clock - temptime;
tel
```

7.3.6 Extending the Fragment with Counters

Extending the framework with counters entails the creation of function *count*. This function gives the number of times a proposition has been true since the beginning of the interval. The equivalent node in Lustre is given in the code below:

```
node count(p,b:bool)returns(count_p_b:int);
 let
  count_p_b = if (after(b) and p)
      then (0->pre(count(p,b)))+1
    else if (after(b)) then (0->pre(count(p,b)))
    else 0;
 tel
```

This can be used to count the number of events by counting the number of rising edges of a proposition. For this reason, we also introduce the *rising* construct which returns true upon a rising edge of a particular proposition.

7.3.7 Defining More Complex Constructs

Writing the formula in the example is quite cumbersome. Therefore, we propose some syntactic sugaring to make the formula more readable. First, we will start with some basic additions which will make it easier to later introduce more complex constructs.

7.3.8 Building Blocks

First, we will introduce the *point* construct which is satisfied if and only if the given proposition is true for the first point in the interval. This will be written as $[P]^0$ and will be defined by $(begin(P) \wedge (len \leq 0))$.

The second addition will be the disjunction of the point ($\lceil P \rceil^0$) and the interval ($\lceil\lceil P \rceil$). This will be written as $\lceil\lceil P \rceil^+$ and will be defined by $\lceil P \rceil^0 \vee \lceil\lceil P \rceil$.

$\lceil\lceil P \rceil\rceil$ by definition is $\lceil\lceil P \rceil \frown \lceil P \rceil^0$. The translation of $\lceil\lceil P \rceil \frown \lceil Q \rceil^0$ into the deterministic fragment is given as $\lceil\lceil P \rceil \wedge end(Q)$. However, we will not use this conversion because this raises the formula to an F fragment (*end* is part of F not G). This is not desirable. Therefore, we give the direct definition of $\lceil\lceil P \rceil\rceil$ as follows: *strict_after*(b) *and always_since*(b, P). This is identical to the definition of $\lceil\lceil P \rceil$ except that we use *always_since* rather than *pre*(*always_since*). In this way, we will also include the last point when P is true.

We define $\lceil\lceil P \rceil\rceil^+$ using its standard definition of $\lceil\lceil P \rceil\rceil \vee \lceil P \rceil^0$.

Proposed Operators

Leads To The *leadsto* operator can be used to define properties where a proposition Q should be true whenever P has been true for more than *delta* time units. This is written as P *leadsto*(*delta*) Q and defined in Lustre by: $age(P) < delta \ or \ Q$ (as given by Gonnord et al.).

Persist The *persist* operator can be used to define properties where a proposition Q has to be true while P has been true for less than *delta* time units. This is written as P *persist*(*delta*) Q and defined in Lustre by: $age(P) \leq 0 \ or \ age(P) > delta \ or \ Q$.

Stable *Stable* is the property that when a given proposition becomes true, it remains true for a given amount of time. This is defined as: $not((false \rightarrow pre(P))\ and\ not(P)\ and\ (false \rightarrow pre(age(P) < delta)))$. This will return false on any instance where P turns false after being true for less than delta. This definition can be modified so that once the property is violated, the acceptor will keep returning false throughout the interval. The necessary modification is the addition of $or\ pre(stable\_output)$ to the previously given statement.

The strong version for stable will not return true if P is true and the *age* is not greater than *delta*. This can be monitored by the acceptor: $repeat(age(P) < delta\ then\ \lceil\lceil P \rceil\rceil^{+}\ then\ \lceil\lceil \neg\, P \rceil\rceil^{+})$.

Always In Lustre there is no way we can run more than one copy of the same acceptor. To check that a property always holds, we would have to start a copy of the acceptor on each state. Thus, *always* cannot be applied on formulae but only on propositions, giving it the equivalent meaning of $\lceil P \rceil$ in duration calculus. *always(P)* means that the property should be true throughout the whole interval. This will be defined by $always = (true \rightarrow pre\ always)\ and\ P$.

Eventually The case of *eventually* is very similar to *always* — it can only be applied to propositions. The property states that the proposition should be at least true for one instant throughout the whole interval. Note that *this* meaning of *eventually* is equivalent to $true \frown \lceil P \rceil \frown true$ in duration calculus. Thus, $eventually(P)$ is defined by $eventually = (false \rightarrow pre\ eventually)\ or\ P$.

Bounded The *bounded* property accepts 3 parameters: a proposition P, a count *alpha* and a duration *delta*. The meaning of such a property is that the proposition cannot be true for more than *alpha* number of times within the given duration of *delta* time units. In order to apply this for the number of events, we apply the *rising* operator on the proposition. This can be useful for example to verify that no more than 5 bad logins occur within any period of 10 minutes. This can be written as: $bounded(rising(badlogin), 5, 10 * 60 * 1000)$. The generated node to monitor such a property requires *alpha* variables to be able to store the timestamp at which each of the events occurred. In this way the node can ensure that the number of allowed events occurred within the allowed time interval.

Iteration

Another issue is that some of the acceptors as defined by Gonnord et al. [45] assume that the *begin* of an acceptor becomes true only once in the lifetime of the acceptor. Iteration of any kind requires the same acceptor to be used over and over again. Hence, this requires an important modification in a number of nodes because these were designed with the assumption that they will only be used once. In the modified version, it is assumed that an interval can be restarted by setting the starting parameter to true more than once.

For example consider the *strict_after* node (which is used in various other nodes):

```
node strict_after(p:bool)returns(strict_after_p:bool);
  let
    strict_after_p = if (false -> pre(p)) then true
      else (false -> pre(strict_after(p)));
  tel

node strict_after(p:bool)returns(strict_after_p:bool);
  let
    strict_after_p = if (p) then false
      else if (false->pre(p)) then true
      else (false -> pre(strict_after(p)));
  tel
```

We also revise our nodes such that for example the *count* node becomes as follows:

```
node count(b,p:bool)returns(count_p_b:int);
  let
    count_p_b = if (b and p) then 1
      else if (b) then 0
      else if (after(b) and p) then (0->pre(count(b,p)))+1
      else if (after(b)) then (0->pre(count(b,p)))
      else 0;
  tel
```

This modification can be further extended such that each acceptor can have another parameter to indicate the end of the interval. In this way the acceptor can return back to the state it was before the first start occurred. However, this was not implemented.

Repeat Sometimes it is desirable to repeat the satisfaction of a formula such that as soon as it is satisfied, it is restarted for the following interval. To this end we added the *repeat* construct. This is simply defined as $repeated\_output = repeated\_acceptor(b\ or\ (false \rightarrow pre(repeated\_output)), params)$. By modifying the *begin* parameter of the acceptor, we will restart the acceptor exactly at the point after an interval was found to satisfy the formula.

7.3.9 Future Work

Further Extensions

The suggested extensions to the QDDC fragment suggested above are far from exhaustive. Other useful extensions may also be required to the Lustre synchronous language. For example, a notable limitation in the current framework is the inability to trigger timer events. This may be very useful in monitoring real-time properties. However, this requires extensive work and is beyond the scope of this work.

Context

With very little modification, the proposed architecture can be extended to represent properties which are verified for individual objects. The same modification suggested for counterexample traces can also be suggested in this case: introducing constructs similar to the *FOREACH* construct in Larva or manual modification of the Larva script output by the translation.

7.4 Larva and Contract Language

7.4.1 Contract Language

With the advancement of service-oriented architecture (SOA), a lot of services are automatically given to customers against a payment. In SOA, this basic idea is extended such that a service can use other services while serving a customer. The interoperability and compositionality issues among services is already a big problem. But security, reliability and trust issues are also very complex. In commerce, there is always a contract between the customer and the service provider, even though it may be implicit. For example, the customer expects to pay the amount written on the price tag of the object being bought; or, if an object just bought does not function, then the customer expects some kind of compensation. It is not easy to represent a contract in logic. However, this would be very beneficial since contracts may be automatically compared against each other and a behaviour may be automatically verified against the agreed contract.

Contract Language ($\mathcal{CL}$) [77] is one of the proposed notations to represent electronic contracts. The advantage is that this language has a logic as a basis — a variant of μ-calculus — thus enabling automatic processing of contracts.

Brief Overview

The basic building blocks of the $\mathcal{CL}$ are obligations, permissions and prohibitions. Obligations are actions which must be performed, permissions are actions which may be performed while prohibitions are action which cannot be performed. One should note that permissions and prohibitions are mutually exclusive because they are the negation of each other. Furthermore, it is assumed that what is obligatory is also permitted. An obligation to perform action α is written as $O(\alpha)$. Similarly, $P(\alpha)$ signifies a permission while $F(\alpha)$ signifies a prohibition to perform action α.

Contrary-to-duty obligations (CTDs) and *contrary-to-prohibitions* (CTPs) are the reparation actions which must be carried out if an obligation is not fulfilled or a prohibition is violated, respectively. An obligation to perform φ in case of the violation of $F(\alpha)$ is written as $F_{\varphi}(\alpha)$.

7.4.2 Monitoring $\mathcal{CL}$ with Larva

Monitoring of contracts is useful because the violation of a contract can thus be detected as soon as it occurs. In this way, the obligations in case of a violation can be enforced.

In yet unpublished work, done by Stephen Fenech [36], $\mathcal{CL}$ has been converted into automata. Thus, it is quite easy for these automata to be converted into DATEs. Once the automata are available, the next step would be to select the system events which trigger the automaton. One should note that DATEs do not support concurrent actions which $\mathcal{CL}$ supports. This is not really a problem because, in a single-threaded system, concurrent method calls are not possible. To show the practicality of monitoring a contract with Larva we provide the example in the following section.

7.4.3 Example

Consider the scenario where we want to ensure that a user logs into a system before being able to request data. Eventually, the user is also allowed to log out. This would be written in $\mathcal{CL}$ as:

$$[\overline{login}^*]F(request) \wedge [1^*][logout][\overline{login}^*]F(request) \wedge F(request)$$

The automaton generated would be as seen in Figure 7.5.

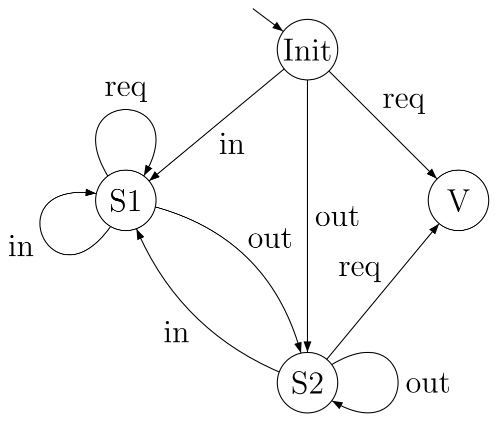


Figure 7.5: Automaton generated from the $\mathcal{CL}$ clause $[\overline{login}^*]F(request) \wedge [1^*][logout][\overline{login}^*]F(request) \wedge F(request)$.

From this automaton we automatically generate the following Larva code:

```
PROPERTY clcontract
{
  STATES
  {
    BAD { V }
    NORMAL { S1 S2 }
    STARTING { Init }
  }
  TRANSITIONS
  {
    Init -> S1 [login]
    Init -> V [request]
    Init -> S2 [logout]
    S1 -> S1 [login]
    S1 -> S1 [request]
    S1 -> S2 [logout]
    S2 -> S2 [logout]
    S2 -> V [request]
    S2 -> S1 [login]
  }
}
```

The final step is to relate the contract actions to method calls. This will need to be done manually since knowledge of the system is required. Let us assume that in order to login into the system a method named *login* is called; in order to logout of the system, a method named *logout* is called; whereas the method *requestItem* is called in order to request information. We specify the events using the code:

```
EVENTS
{
login = {*.login()}
logout= {*.logout()}
request= {*.requestItem()}
}
```

Using aspect-oriented programming, the monitoring framework will be notified when the methods *login*, *logout* or *requestItem* are called on any object. This would trigger the automaton which, in turn, would monitor the contract which we specified.

7.5 Conclusion

DATEs are not always the most appropriate logic to express certain properties. Thus, in this chapter, we have considered other logics which can be translated into DATEs. A highly expressive logic is duration calculus which is very useful to express real-time properties. The full duration calculus cannot be translated into DATEs but a subset called counterexample traces can. For this reason, we gave the translation of counterexample traces into DATEs and proved it correct. Using this translation, one can still use Larva as the underlying framework and, at the same time, express properties in counterexample traces. Another useful subset of duration calculus is QDDC. This has also been shown to be translatable into Larva by going through Lustre. The consequence is that both QDDC and Lustre can be used to write properties which can be monitored in Larva. A notable advantage of Lustre is that its nodes keep modularity which allows quite complex properties to be written clearly. Furthermore, Lustre has the advantage that upperbounds for memory and temporal overheads can be calculated at compile time.

For users which are used to duration calculus, QDDC may be a good option, especially by using the syntactic sugaring suggested in this chapter.

Finally, we have shown $\mathcal{CL}$ — a language to describe electronic contracts — to be translatable into Larva automata. Using this translation, Larva can also be used to monitor contracts.

8. Slowdown and Speedup Truth Preservation

8.1 Introduction

Runtime verification of real-time properties introduces new challenges since the system is usually slowed down by the parallel execution of the monitor, affecting the verification result. The ideal solution for this problem is to use monitors which are so lightweight, that the overhead they introduce is negligible. This is, however, virtually impossible. Thus, the possibility of monitoring real-time properties without affecting their satisfaction or violation seems to be very remote. However, we claim to have found a compromise by which we can identify a class of properties which are not affected by the overhead of monitoring. More specifically, if a property holds for a particular execution of a system, it will also hold in a slowed down version of the same execution — the property preserves truth. For this purpose, we analyse duration calculus to identify such a subset of *slowdown truth preserving properties*. Conversely, there is also a subset of *speedup truth preserving* properties. For example, the property "no more than 3 bad logins are allowed in 5 minutes" is a slowdown truth preserving property, i.e. if the system is slowed down, the number of bad logins may only decrease (not increase). Thus, if the property is satisfied, it will still be satisfied when overheads are introduced through monitoring.

In the following section, we will give a whole theory of slowdown (and speedup, respectively) truth preservation. Section 8.3 explains the applications where this theory may be useful. The last section concludes the chapter.

8.2 Duration Calculus and Slowdown Truth Preservation

In this section, we define the notion of slowdown and speedup truth preservation. We show that many interesting properties (in duration calculus) are slowdown truth preserving (SDTP) or speedup truth preserving (SUTP). For example, consider the formula: $\neg \Diamond(\lceil Danger \rceil \wedge \ell > \delta \frown \lceil \neg\, Alarm \rceil)$. This means that it is never the case that, after the *Danger* signal has been on for more than δ time units, the *Alarm* signal is off. This property is speedup truth preserving, i.e. if the property holds on a slow system, the *Alarm* can only turn on earlier in the speeded-up system. Thus, the property will still hold. We also show that the negation of a slowdown truth preserving property is speedup truth preserving and vice-versa. Identifying slowdown truth preserving properties for duration calculus is very useful since, as we will see in next section, we know how to translate a fragment of duration calculus into phase event automata and then into Larva.

8.2.1 Slowdown and Speedup Truth Preservation

Slowing down a system means that events happen at a slower rate. The behaviour of the slower system will be identical to the original one, except that time will be continuously transformed in a monotonic fashion. In a similar way, speeding up a system means that the behaviour remains the same but happens in a shorter time frame. In this subsection we will give a mathematical framework for slowdown and speedup properties. Once this ground work is ready, in the next subsection we will prove a number of formulas to be slowdown or speedup truth preserving.

We will start our definitions by giving the notion of a time transformation function which transforms the time at which events occur. Time transforms possess a number of useful properties which allow us to prove interesting properties later on.

Definition 8.2.1. A continuous total continuous function $s \in \mathbb{T} \rightarrow \mathbb{T}$ is said to be a time transformation ($s \in \mathcal{T}$) if (i) $s(0) = 0$; (ii) $\lim_{t\rightarrow\infty} s(t) = \infty$; (iii) s is monotonic ($t_1 < t_2 \Rightarrow s(t_1) < s(t_2)$).

The simplest time transform is the identity function which, given time t, returns t as output: $s(t) = t$.

Given a time transformation s, and an interpretation I, we define the transformed interpretation of I with respect to s, written I_s, as follows:

$$\forall\, P,\ t \cdot I_s(P)(s(t)) = I(P)(t)$$

A useful property of the identity function is that when applied to an interpretation, the latter remains unchanged. This property will by useful in other proofs.

Proposition 8.2.1. If the identity function is applied to an interpretation I, the resulting interpretation I_{id} is equal to I.

Proof. This follows easily by using the definition of the identity function $s(t) = t$:

$$\forall\, P,\ t \cdot I_{id}(P)(t) = I(P)(t)$$

Thus,

$$I_{id} = I$$

□

In a number of proofs, we require functional composition of time transforms, i.e. that more than one time transform is applied on an interpretation. This is particularly useful when using the equivalence $s_\circ s^{-1} = id$.

Proposition 8.2.2. Applying time transform f on an interpretation, followed by the application of time transform g is equivalent to applying the functional composition $g_\circ f$:

$$(I_f)_g = I_{g_\circ f}$$

A naïve way of defining a time transform to be a time-stretch, is to insist that $s(t) \geq t$. However, this would only guarantee that all event timings of an interpretation I occur earlier than those of the I_s — this is not what we require, since it does not guarantee that the intervals between events are always longer in the slow interpretation than the fast counterpart[1].

Definition 8.2.2. A time transformation $s \in \mathcal{T}$ is said to be a time-stretch ($s \in \overleftrightarrow{\mathcal{T}}$) if it is monotonic on intervals: $s(t_2) - s(t_1) \geq t_2 - t_1$ (for $t_1 < t_2$). Similarly, it is said to be a time-compression ($s \in \overset{\rightarrow\leftarrow}{\mathcal{T}}$) if it is anti-monotonic on intervals: $s(t_2) - s(t_1) \leq t_2 - t_1$ (for $t_1 < t_2$).

By the above definition of a time-stretch, we are also adhering to the naïve definition $s(t) \geq t$. The proof is as follows:

Proposition 8.2.3. The constraint $s(t_2) - s(t_1) \geq t_2 - t_1$ (for $t_1 < t_2$) implies that $s(t) > t$

Proof. Let $t_2 = t$ and $t_1 = 0$.

$$\begin{aligned} & s(t_2) - s(t_1) \geq t_2 - t_1 \\ \Longrightarrow\ & \text{By substitution} \\ & s(t) - s(0) \geq t - 0 \\ \Longrightarrow\ & \text{By definition of a time transform } s(0) = 0 \\ & s(t) - 0 \geq t - 0 \\ \Longrightarrow\ & \text{Basic calculus} \\ & s(t) \geq t \end{aligned}$$

[1]Consider the case when the first two changes of a boolean variable X occur at times 5 and 10 under an interpretation I, but at times 9 and 11 under I_s. All other events occur at the same time in the two interpretations. Although all events of I' occur no later than the events in I, if one looks at the time between the first and second event, it is actually smaller in the case of I. If causality of events is seen to start building up from the previous event, we need to look at lengthening intervals between events, not at delays in the events on an absolute time line — this requires interval-monotonicity.

□

To show the relationship between time-stretch functions and time-compression functions, we require the inverse of these functions and thus, first we need to show that time transforms are bijective. To prove that a function is bijective, first we require to prove that it is injective and secondly, that it is surjective.

Lemma 8.2.1. Time transforms are injective.

Proof. Assume that $f(t_1) = f(t_2)$. The three possibilities are that either $t_1 < t_2$, $t_1 > t_2$, or $t_1 = t_2$. However, by monotonicity of a time transform, $t_1 < t_2 \Rightarrow f(t_1) < f(t_2)$. This contradicts $f(t_1) = f(t_2)$ and thus, the possibility of $t_1 < t_2$ is eliminated. Similarly, we can show that $t_1 \not> t_2$. Hence, $t_1 = t_2$. □

Lemma 8.2.2. Time transforms are surjective.

Proof. Given the limits $f(0) = 0$, $\lim_{t\to\infty} s(t) = \infty$, and continuity of function f, surjectivity follows as an application of the intermediate value theorem [50]. □

Corollary 8.2.1. Time transforms are bijective. This follows easily because we have already shown that time transforms are injective (Lemma 8.2.1) and surjective (Lemma 8.2.2).

Proposition 8.2.4. The inverse of every time-stretch transformation is a time-compress transformation:

$$s \in \overleftrightarrow{\mathbb{T}} \Rightarrow s^{-1} \in \overset{\rightarrow\!\leftarrow}{\mathbb{T}}$$

The inverse of every time-compress transformation is a time-stretch transformation:

$$s \in \overset{\rightarrow\!\leftarrow}{\mathbb{T}} \Rightarrow s^{-1} \in \overleftrightarrow{\mathbb{T}}$$

Proof. To prove $s \in \overleftrightarrow{\mathbb{T}} \Rightarrow s^{-1} \in \overset{\rightarrow\!\leftarrow}{\mathbb{T}}$ we will start by $s \in \overleftrightarrow{\mathbb{T}}$ and arrive at $s^{-1} \in \overset{\rightarrow\!\leftarrow}{\mathbb{T}}$. A time compression function is a time transform. Thus, by the definition of a time transform (Definition 8.2.1) the following conditions must hold: (i) $s^{-1}(0) = 0$; (ii) $\lim_{t\to\infty} s^{-1}(t) = \infty$; and (iii) s^{-1} is monotonic ($t_1 < t_2 \Rightarrow s^{-1}(t_1) < s^{-1}(t_2)$). By definition of a time compression function (Definition 8.2.2), the condition (iv) $s^{-1}(t_2) - s^{-1}(t_1) \leq t_2 - t_1$ (for $t_1 < t_2$) must hold.

(i) By Definition 8.2.1, $s(0) = 0$ and thus, $s^{-1}(0) = 0$.

(ii) By Definition 8.2.1, $\lim_{t\to\infty} s(t) = \infty$ and thus, $\lim_{t\to\infty} s^{-1}(t) = \infty$.

(iii) We will prove monotonicity by contradiction. Initially, we will assume that $t_1 < t_2 \Rightarrow s^{-1}(t_1) \geq s^{-1}(t_2)$

$$\begin{aligned}
& s^{-1}(t_1) \geq s^{-1}(t_2) \\
\Longrightarrow\ & \text{Applying the function } s \text{ on both sides — } s \text{ is monotonic} \\
& s(s^{-1}(t_1)) \geq s(s^{-1}(t_2)) \\
\Longrightarrow\ & \text{Functional composition and } id = s_\circ s^{-1} \\
& id(t_1) \geq id(t_2) \\
\Longrightarrow\ & \text{The id function returns its input} \\
& t_1 \geq t_2 \\
\Longrightarrow\ & \text{Contradiction } (t_1 < t_2) \text{ — } s^{-1} \text{ is monotonic} \\
& s^{-1}(t_1) < s^{-1}(t_2)
\end{aligned}$$

(iv) Let $t_1 = s(k_1)$ and $t_2 = s(k_2)$. Thus, since s is bijective (Corollary 8.2.1), $k_1 = s^{-1}(t_1)$ and $k_2 = s^{-1}(t_2)$.

$$\begin{aligned}
& s \in \overleftrightarrow{\mathcal{T}} \\
\Longrightarrow\ & \text{By definition of a time-stretch (Definition 8.2.2)} \\
& s(k_2) - s(k_1) \geq k_2 - k_1 \text{ (for } k_1 < k_2) \\
\Longrightarrow\ & \text{By substitution} \\
& t_2 - t_1 \geq s^{-1}(t_2) - s^{-1}(t_1) \text{(for } k_1 < k_2) \\
\Longrightarrow\ & \text{By definition of a time-compression} \\
& s^{-1} \in \overset{\rightarrow\!\leftarrow}{\mathcal{T}}
\end{aligned}$$

□

The proof is similar for $s \in \overset{\rightarrow\!\leftarrow}{\mathcal{T}} \Rightarrow s^{-1} \in \overleftrightarrow{\mathcal{T}}$.

The integral operator in duration calculus is very useful and serves as a basis for other operators. The following proposition will be used later on when we prove truth preservation properties on the integral operator.

Proposition 8.2.5. Given a time-stretch s, $\int_{s(b)}^{s(e)} \alpha(t)\,dt \geq \int_b^e \alpha(s(t))\,dt$. Similarly, given a time-compression f, $\int_{f(b)}^{f(e)} \alpha(t)\,dt \leq \int_b^e \alpha(f(t))\,dt$.

Proof. The informal argument of the proof is the following:

By the finite variability property of α, there are only a finite number of discontinuous points. Thus, we can choose a sequence $b, t_1, t_2, \ldots, t_n, e$ such that the function α is constant from $\alpha(s(b))$ till $\alpha(s(t_1))$, from $\alpha(s(t_1))$ till $\alpha(s(t_2))$ and so on. The integrals can be divided into small parts as follows:

$$\begin{aligned}
\int_{s(b)}^{s(e)} \alpha(t)\,dt &= \int_{s(b)}^{s(t_1)} \alpha(t)\,dt + \int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt + \ldots + \int_{s(t_n)}^{s(e)} \alpha(t)\,dt \\
\int_{b}^{e} \alpha(s(t))\,dt &= \int_{b}^{t_1} \alpha(s(t))\,dt + \int_{t_1}^{t_2} \alpha(s(t))\,dt + \ldots + \int_{t_n}^{e} \alpha(s(t))\,dt
\end{aligned}$$

A discontinuous function between any two points $s(t_1)$ and $s(t_2)$, either continuously evaluates to 1, or continuously evaluates to 0. Let us consider the first case such that the value of α between the limits evaluates to 0:

$$\begin{aligned}
\int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt &= 0 \\
\int_{t_1}^{t_2} \alpha(s(t))\,dt &= 0
\end{aligned}$$

It is clear that $\int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt \geq \int_{t_1}^{t_2} \alpha(s(t))\,dt$. Now let us consider the second case such that between the points $s(t_1)$ and $s(t_2)$, α evaluates to 1:

$$\begin{aligned}
\int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt &= s(t_2) - s(t_1) \\
\int_{t_1}^{t_2} \alpha(s(t))\,dt &= t_2 - t_1
\end{aligned}$$

By definition of a time-stretch $s(t_2) - s(t_1) > t_2 - t_1$, it follows that $\int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt \geq \int_{t_1}^{t_2} \alpha(s(t))\,dt$. This leads us to conclude that, since both possibilities (α evaluating to 1 and α evaluating to 0) imply that $\int_{s(t_1)}^{s(t_2)} \alpha(t)\,dt \geq \int_{t_1}^{t_2} \alpha(s(t))\,dt$, then this holds for all parts of the integrals. Thus, by considering the addition of all the parts of both integrals, we can conclude that $\int_{s(b)}^{s(e)} \alpha(t)\,dt \geq \int_{b}^{e} \alpha(s(t))\,dt$.

The proof is similar in the case of a time-compression.

$\square$

The previous propositions and definitions provide the ground work for the proofs that will follow. The next step is to provide formal definitions of the terms which will be used in the rest of this chapter.

Although we have given an informal explanation of slowdown truth preservation, the following definitions give the exact definition which will be used in the proofs. In the first definition we explain the meaning of truth preservation.

Definition 8.2.3. A duration formula D is said to be slowdown (speedup) truth preserving *sdtp*(D) (*sutp*(D)) if, for any interpretation I on which D holds for all finite prefixes of time, D also holds for all finite prefixes of time under any slowdown (speedup) of the interpretation I_s (I_f):

$$\mathit{sdtp}(D) \overset{\text{def}}{=} \forall\, s, I \;\cdot\; I \models D \Rightarrow I_s \models D$$
$$\mathit{sutp}(D) \overset{\text{def}}{=} \forall\, f, I \;\cdot\; I \models D \Rightarrow I_f \models D$$

The counterpart of truth preserving is falsity preservation, i.e. that a formula which does not hold, remains unsatisfied in the modified interpretation.

Definition 8.2.4. A duration formula D is said to be slowdown (speedup) false preserving *sdfp*(D) (*sufp*(D)) if, for any interpretation I on which D is violated on all finite prefixes of time, D is also violated on all finite prefixes of time under any slowdown (speedup) of the interpretation I_s (I_f):

$$\mathit{sdfp}(D) \overset{\text{def}}{=} \forall\, s, I \;\cdot\; I \not\models D \Rightarrow I_s \not\models D$$
$$\mathit{sufp}(D) \overset{\text{def}}{=} \forall\, f, I \;\cdot\; I \not\models D \Rightarrow I_f \not\models D$$

Combining the two previous definitions, a formula is slowdown or speedup invariant if it preserves both the truth and the falsity. Such formulas can be very useful when monitoring real-time properties.

Definition 8.2.5. A duration formula D is said to be slowdown (speedup) invariant *sdi*(D) (*sui*(D)) if *sdtp*(D) and *sdfp*(D) (*sutp*(D) and *sufp*(D)).

$$\mathit{sdi}(D) \overset{\text{def}}{=} \mathit{sdtp}(D) \text{ and } \mathit{sdfp}(D)$$
$$\mathit{sui}(D) \overset{\text{def}}{=} \mathit{sutp}(D) \text{ and } \mathit{sufp}(D)$$

So far, we have considered the satisfaction of formulas on all the prefixes of an interpretation. This can be more generalised so that we consider the satisfaction of a formula on all the subintervals of an interpretation. Formulae which preserve truth under interval stretching will be called *interval-stretch truth preserving*, and *interval-compress truth preserving* if they preserve truth under compression of intervals. This is more powerful and it can be easily shown (see Theorem 8.2.1) that prefixes are simply a particular subset of subintervals. Apart from proving these properties on formulae for their own sake, they are also useful because the properties proved on intervals, also apply for prefixes. The following definition explains what it means for a formula to preserve truth on stretched intervals.

Definition 8.2.6. A duration formula D is said to be interval-stretch (interval-compress) truth preserving *isdtp*(D) (*isutp*(D)) if, for all interpretations I on which D is satisfied on all subintervals, D also holds for all subintervals under any slowdown (speedup) of the interpretation I_s (I_f)

$$\mathit{isdtp}(D) \overset{\text{def}}{=} \forall\, s, I, b, e \;\cdot\; I \models_{[b,e]} D \Rightarrow I_s \models_{[s(b),s(e)]} D$$
$$\mathit{isutp}(D) \overset{\text{def}}{=} \forall\, f, I, b, e \;\cdot\; I \models_{[b,e]} D \Rightarrow I_f \models_{[f(b),f(e)]} D$$

After considering truth preservation, we similarly consider the preservation of falsity.

Definition 8.2.7. A duration formula D is said to be interval-stretch (interval-compress) false preserving $\mathit{isdfp}(D)$ ($\mathit{isufp}(D)$) if, for any interpretation I on which D is violated on all subintervals, D is also violated on all subintervals under any slowdown (speedup) of the interpretation I_s (I_f):

$$\mathit{isdfp}(D) \overset{\text{def}}{=} \forall\, s, I, b, e \;\cdot\; I \not\models_{[b,e]} D \Rightarrow I_s \not\models_{[s(b),s(e)]} D$$
$$\mathit{isufp}(D) \overset{\text{def}}{=} \forall\, f, I, b, e \;\cdot\; I \not\models_{[b,e]} D \Rightarrow I_f \not\models_{[f(b),f(e)]} D$$

When we combine the above two definitions, we get a formula which is interval-stretch (or interval-compress) invariant, i.e. we preserve both truth and falsity.

Definition 8.2.8. A duration formula D is said to be interval-stretch (interval-compress) invariant $\mathit{isdi}(D)$ ($\mathit{isui}(D)$) if $\mathit{isdtp}(D)$ and $\mathit{isdfp}(D)$ ($\mathit{isutp}(D)$ and $\mathit{isufp}(D)$).

$$\mathit{isdi}(D) \overset{\text{def}}{=} \mathit{isdtp}(D) \text{ and } \mathit{isdfp}(D)$$
$$\mathit{isui}(D) \overset{\text{def}}{=} \mathit{isutp}(D) \text{ and } \mathit{isufp}(D)$$

As explained before the definitions of interval-based properties, the result of interval-based properties is more general than prefix-based properties. The following theorem proves that an interval-stretch truth preserving formula is also slowdown truth preserving.

Theorem 8.2.1. A duration formula D which is interval-stretch (interval-compress) truth preserving $\mathit{isdtp}(D)$ ($\mathit{isutp}(D)$), is also slowdown truth preserving $\mathit{sdtp}(D)$ ($\mathit{sutp}(D)$):

$$\mathit{isdtp}(D) \Rightarrow \mathit{sdtp}(D)$$

Proof. In order to prove $\mathit{sdtp}(D)$ we have to prove the implication:

$$I \models D \Rightarrow I_s \models D$$

$$\begin{array}{ll}
 & I \models D \\
\Longrightarrow & \text{By definition of } \models \\
 & \forall\, t : \mathbb{T} \;\cdot\; I \models_{[0,t]} D \\
\Longrightarrow & \mathit{isdtp}(D) \\
 & \forall\, t, s : \mathbb{T} \;\cdot\; I_s \models_{[s(0),s(t)]} D \\
\Longrightarrow & s(0) = 0 \\
 & \forall\, t, s : \mathbb{T} \;\cdot\; I_s \models_{[0,s(t)]} D \\
\Longrightarrow & s \text{ is total} \\
 & \forall\, t', s : \mathbb{T} \;\cdot\; I_s \models_{[0,t']} D \\
\Longrightarrow & \text{By definition of } \models \\
 & I_s \models D
\end{array}$$

□

Using the above theorem, the operators and duration formulas shown to be interval-stretching (respectively interval-compression) truth (respectively false) preserving are also slowdown (respectively speedup) truth (respectively false) preserving.

The following two propositions show the relationship between preservation of truth and falsity. Informally, a formula which preserves truth on a stretched interval, preserves falsity on a compressed interval. This conclusion will be used later on to show the relationship between the negation and truth preservation.

Proposition 8.2.6. Every interval-stretch truth preserving formula $\mathit{isdtp}(D)$, is also interval-compress false preserving $\mathit{isufp}(D)$ and vice-versa:

$$\mathit{isdtp}(D) \Leftrightarrow \mathit{isufp}(D)$$

Proof. First we will show that, assuming $\mathit{isdtp}(D)$, $\mathit{isufp}(D)$ is true. In other words, using the definition of $\mathit{isufp}(D)$ we have to show that:

$$I \not\models_{[b,e]} D \Rightarrow I_f \not\models_{[f(b),f(e)]} D$$

$$\begin{array}{ll}
 & I \not\models_{[b,e]} D \\
\Longrightarrow & \text{Proposition 8.2.1} \\
 & I_{id} \not\models_{[b,e]} D \\
\Longrightarrow & f_\circ f^{-1} = id \ (f \text{ being a time compress function}) \\
 & I_{f_\circ f^{-1}} \not\models_{[f_\circ f^{-1}(b), f_\circ f^{-1}(e)]} D \\
\Longrightarrow & \text{Function composition (Proposition 8.2.2)} \\
 & (I_f)_{f^{-1}} \not\models_{[f^{-1}(f(b)), f^{-1}(f(e))]} D \\
\Longrightarrow & \mathit{isdtp}(D),\ f^{-1} \text{ is slowdown (Proposition 8.2.4)}, \\
 & \text{contrapositive } (I_s \not\models_{[s(b),s(e)]} D \Rightarrow I \not\models_{[b,e]} D) \\
 & I_f \not\models_{[f(b),f(e)]} D
\end{array}$$

For the other direction, we will show that, assuming $\mathit{isufp}(D)$, $\mathit{isdtp}(D)$ is true. In other words, using the definition of $\mathit{isdtp}(D)$ we have to show that:

$$I \models_{[b,e]} D \Rightarrow I_s \models_{[s(b),s(e)]} D$$

$$\begin{array}{ll}
 & I \models_{[b,e]} D \\
\Longrightarrow & \text{Proposition 8.2.1} \\
 & I_{id} \models_{[b,e]} D \\
\Longrightarrow & s_{\circ}s^{-1} = id \ (s \text{ being a time stretch function}) \\
 & I_{s_{\circ}s^{-1}} \models_{[s_{\circ}s^{-1}(b), s_{\circ}s^{-1}(e)]} D \\
\Longrightarrow & \text{Function composition (Proposition 8.2.2)} \\
 & (I_{s^{-1}})_s \models_{[s^{-1}(s(b)), s^{-1}(s(e))]} D \\
\Longrightarrow & \mathit{isufp}(D),\ s^{-1} \text{ is speedup (Proposition 8.2.4),} \\
 & \text{contrapositive } (I_f \not\models_{[f(b),f(e)]} D \Rightarrow I \models_{[b,e]} D) \\
 & I_s \models_{[s(b),s(e)]} D
\end{array}$$

□

The next proposition is very similar to the previous one. It states that a formula which is interval-compress truth preserving is also interval-stretch false preserving and vice-versa. We do not give the actual proof of this proposition because it is very similar to that of the previous proposition.

Proposition 8.2.7. Every interval-compress truth preserving formula $\mathit{isutp}(D)$, is also interval-stretch false preserving $\mathit{isdfp}(D)$ and vice-versa:

$$\mathit{isutp}(D) \Leftrightarrow \mathit{isdfp}(D)$$

Proof. Proof is similar to that for Proposition 8.2.6.

□

An immediate consequence of the two previous propositions is that a formula which is both interval-compress truth preserving ($\mathit{isutp}(D)$) and interval-compress false preserving ($\mathit{isufp}(D)$) — interval-compress invariant ($\mathit{isui}(D)$) — is also interval-stretch invariant ($\mathit{isdi}(D)$) and vice-versa.

Corollary 8.2.2. A formula which is interval-compress invariant is also interval-stretch invariant and vice-versa:

$$\mathit{isui}(D) \Leftrightarrow \mathit{isdi}(D)$$

Proof.

$$\begin{array}{rl} & isui(D) \\ \iff & \text{By definition of } isui(D) \\ & isutp(D) \wedge isufp(D) \\ \iff & \text{Proposition 8.2.6, 8.2.7} \\ & isdfp(D) \wedge isdtp(D) \\ \iff & \text{By definition of } isdi(D) \\ & isdi(D) \end{array}$$

□

An interesting theorem which follows easily from the above regards the relationship of truth preservation and negated formulae. In fact, the negation of a formula which is interval-stretch truth preserving, is interval-stretch false preserving. This similarly applies to interval compressions. Intuitively, the truth preservation of a formula is equivalent to the falsity preservation of its negation.

Theorem 8.2.2. The negation of a formula which is interval-stretch truth preserving, is interval-stretch false preserving:

$$isdtp(D) \Leftrightarrow isdfp(\neg\, D)$$

Proof. In order to show that $isdfp(\neg\, D)$ (assuming $isdtp(D)$), we have to prove the following implication:

$$I \not\models_{[b,e]} \neg\, D \Rightarrow I_s \not\models_{[s(b),s(e)]} \neg\, D$$

$$\begin{array}{rl} & I \not\models_{[b,e]} \neg\, D \\ \Longrightarrow & \text{definition of } \neg\, D \text{ and } \not\models \\ & I \models_{[b,e]} D \\ \Longrightarrow & isdtp(D) \\ & I_s \models_{[s(b),s(e)]} D \\ \Longrightarrow & \text{definition of } \neg\, D \text{ and } \models \\ & I_s \not\models_{[s(b),s(e)]} \neg\, D \end{array}$$

To show that $isdtp(D)$ (assuming $isdfp(\neg\, D)$), we have to prove the following implication:

$$I \models_{[b,e]} D \Rightarrow I_s \models_{[s(b),s(e)]} D$$

The proof is similar to the one above.

□

Similarly, the negation of a formula which is interval-compress truth preserving, is interval-compress false preserving:

$$isutp(D) \Leftrightarrow isufp(\neg\ D)$$

The proof is similar to the one above.

Using the relationship between truth preservation and falsity preservation, the previous theorem can be used to show further interesting results. This time, we will prove that the negation of an interval-stretch truth preserving formula is interval-compress truth preserving. Similarly, the negation of an interval-compress truth preserving formula is interval-stretch truth preserving. This may not be immediately clear. However, using the previous theorem, Proposition 8.2.6 and Proposition 8.2.7, the proof follows easily. This result is important because it shows that the negation transforms an interval-stretch truth preserving formula into an interval-compress truth preserving formula. Recall that this can also be applied to prefixes such that the negation transforms a slowdown truth preserving formula into a speedup truth preserving formula. This result is also very useful for proving other important operators later on such as the $\Box$ operator.

Theorem 8.2.3. The negation of an interval-stretch truth preserving formula is interval-compress truth preserving:

$$isdtp(D) \Rightarrow isutp(\neg\ D)$$

Proof.

$$\begin{array}{ll} & isdtp(D) \\ \Longrightarrow & \text{Proposition 8.2.6} \\ & isufp(D) \\ \Longrightarrow & \text{Theorem 8.2.2} \\ & isutp(\neg\ D) \end{array}$$

Similarly, the negation of an interval-compress truth preserving formula is interval-stretch truth preserving:

$$isutp(D) \Rightarrow isdtp(\neg\ D)$$

The proof is similar to the one above.

$\Box$

A corollary of the above results is that since an interval-compress invariant formula preserves both truth and falsity, then its negation also preserves truth and falsity. This also applies for interval-stretches. This follows easily from the fact that the negation of a formula which preserves truth, preserves falsity and vice-versa.

Corollary 8.2.3. The negation of an interval-compress invariant formula is also interval-compress invariant:

$$isui(D) \Leftrightarrow isui(\neg\ D)$$

Proof.

$$\begin{array}{ll} & isui(D) \\ \Longleftrightarrow & \text{By definition of } isui(D) \\ & isutp(D) \wedge isufp(D) \\ \Longleftrightarrow & \text{Theorem 8.2.2 and Theorem 8.2.3} \\ & isufp(\neg\, D) \wedge isutp(\neg\, D) \\ \Longleftrightarrow & \text{By definition of } isui(\neg\, D) \\ & isui(\neg\, D) \end{array}$$

□

Similarly, the negation of an interval-stretch invariant formula is also interval-stretch invariant:

$$isdi(D) \Leftrightarrow isdi(\neg\, D)$$

The proof is similar to the above.

This subsection has provided the necessary ground work for the actual proofs considering duration calculus fragments. The following subsection will consider parts of duration calculus and show which of the properties defined above hold on each fragment.

8.2.2 Duration Calculus Fragments which Preserve Truth

The first duration calculus formula which we will consider is the most simple one — *true*. This formula is satisfied by all the intervals and will be written as tt. A similar, yet opposite formula, is the formula *ff* which is violated by any interval.

Definition 8.2.9. Let tt be the duration formula which is satisfied by all intervals:

$$tt \overset{\text{def}}{=} \ell \geq 0$$

Let *ff* be the duration formula which is not satisfied by any interval:

$$ff \overset{\text{def}}{=} \neg\, tt$$

Intuitively, it is clear that the above formulae are not affected by time stretches or compressions. These formulae will be useful as building blocks for other formulae. The following theorem formally shows that tt and *ff* are not affected by time transforms.

Theorem 8.2.4. tt is interval-stretch truth preserving: $isdtp(tt)$ and interval-compress truth preserving: $isutp(tt)$. Similarly, *ff* is interval-stretch truth preserving: $isdtp(ff)$ and interval-compress truth preserving: $isutp(ff)$.

Proof.

$$
\begin{array}{rl}
 & I \models_{[b,e]} tt \\
\Longrightarrow & \text{definition of } tt \\
 & I_s \models_{[s(b),s(e)]} tt \\
\Longrightarrow & \text{definition of } \mathit{isdtp}(tt) \\
 & \mathit{isdtp}(tt)
\end{array}
$$

□

Similarly for *isutp*(tt), *isdtp*(ff), and *isutp*(ff).

An important building block in duration calculus formulae is the integral operator. The following theorem shows that the inequality $\int P > c$ is interval-stretch truth preserving. Intuitively, stretch an interval means that the length of time a variable is true increases. On the same lines, it is shown that the inequality $\int P < c$ is interval-compress truth preserving.

Theorem 8.2.5. The inequality $\int P > c$ is interval-stretch truth preserving:

$$\forall\, s : \overleftrightarrow{\mathcal{T}} \cdot I \models_{[b,e]} \int P > c \Rightarrow I_s \models_{[s(b),s(e)]} \int P > c$$

Proof.

$$
\begin{array}{rl}
 & I \models_{[b,e]} \displaystyle\int P > c \\
\Longrightarrow & \text{definition of } \int \\
 & \displaystyle\int_b^e I(P)(t)\,dt > c \\
\Longrightarrow & \text{definition of } I_s \\
 & \displaystyle\int_b^e I_s(P)(s(t))\,dt > c \\
\Longrightarrow & \text{Proposition 8.2.5} \\
 & \displaystyle\int_{s(b)}^{s(e)} I_s(P)(t)\,dt \geq \int_b^e I_s(P)(s(t))\,dt \\
\Longrightarrow & \text{transitivity of} > \\
 & \displaystyle\int_{s(b)}^{s(e)} I_s(P)(t)\,dt > c \\
\Longrightarrow & \text{definition of } \int \\
 & I_s \models_{[s(b),s(e)]} \displaystyle\int P > c
\end{array}
$$

□

The inequality $\int P < c$ is interval-compress truth preserving:

$$\forall f : \overset{\rightarrow\!\times\!\leftarrow}{\mathcal{T}} \cdot I \models_{[b,e]} \int P < c \Rightarrow I_f \models_{[f(b),f(e)]} \int P < c$$

The proof is similar to the previous one.

The next operator to be considered is *almost everywhere* operator $\lceil - \rceil$. It is shown in the following theorem that this operator is both interval-stretch and interval-compress truth preserving.

Theorem 8.2.6. $\lceil P \rceil$ is interval-stretch truth preserving: *isdtp*$(\lceil P \rceil)$:

$$I \models_{[b,e]} \lceil P \rceil \Rightarrow \forall s : \overleftrightarrow{\mathcal{T}} \cdot I_s \models_{[s(b),s(e)]} \lceil P \rceil$$

Proof.

$$\begin{array}{ll}
& I \models_{[b,e]} \lceil P \rceil \\
\Longrightarrow & \text{definition of } \lceil - \rceil \\
& \displaystyle\int_b^e I(P)(t)dt = e - b \wedge e > b \\
\Longrightarrow & \text{definition of } I_s \text{ and monotonicity of time transforms} \\
& \displaystyle\int_b^e I_s(P)(s(t))dt = e - b \wedge s(e) > s(b) \\
\Longrightarrow & \text{Basic Calculus} \\
& \displaystyle\int_{s(b)}^{s(e)} I_s(P)(t)dt = s(e) - s(b) \wedge s(e) > s(b) \\
\Longrightarrow & \text{definition of } \lceil - \rceil \\
& I_s \models_{[s(b),s(e)]} \lceil P \rceil
\end{array}$$

□

A similar proof can be used to show that $\lceil P \rceil$ is interval-compress truth preserving: *isutp*$(\lceil P \rceil)$.

A very important connector in duration calculus is the *chop* operator. Showing that such a connector is interval-stretch truth preserving means that if the chop-connected formulae are interval-stretch truth preserving, this is further preserved in the whole formula.

Theorem 8.2.7. If duration formulae D and E are interval-stretch truth preserving *isdtp*(D) and *isdtp*(E), then $D \frown E$ is also interval-stretch truth preserving *isdtp*$(D \frown E)$:

$$I \models_{[b,e]} D \frown E \Rightarrow I_s \models_{[s(b),s(e)]} D \frown E$$

Proof.

$$\begin{array}{ll}
& I \vDash_{[b,e]} D \frown E \\
\Longrightarrow & \text{By definition of } \frown \\
& \exists\, m : [b, e] \;\cdot\; I \vDash_{[b,m]} D \wedge I \vDash_{[m,e]} E \\
\Longrightarrow & \mathit{isdtp}(D) \text{ and } \mathit{isdtp}(E) \\
& \exists\, m : [b, e] \;\cdot\; I_s \vDash_{[s(b),s(m)]} D \wedge I \vDash_{[s(m),s(e)]} E \\
\Longrightarrow & \text{By monotonicity of } f \\
& \exists\, m' : [s(b), s(e)] \;\cdot\; I_s \vDash_{[s(b),m']} D \wedge I \vDash_{[m',s(e)]} E \\
\Longrightarrow & \text{By definition of } \frown \\
& I_s \vDash_{[s(b),s(e)]} D \frown E
\end{array}$$

□

A similar proof can be used to show that if duration formulae D and E are interval-compress truth preserving $\mathit{isutp}(D)$ and $\mathit{isutp}(E)$, then $D \frown E$ is also interval-compress truth preserving $\mathit{isutp}(D \frown E)$.

Using the chop and the duration formula tt, we can construct the operator *eventually* D: $\Diamond D$. Since we have already proved these constructs to be truth preserving, the proof of $\Diamond D$ follows easily.

Theorem 8.2.8. If a duration formula D is interval-stretch truth preserving $\mathit{isdtp}(D)$, then $\Diamond D$ is also interval-stretch truth preserving $\mathit{isdtp}(\Diamond D)$:

$$I \vDash_{[b,e]} \Diamond D \Rightarrow I_s \vDash_{[s(b),s(e)]} \Diamond D$$

Proof.

$$\begin{array}{ll}
& I \vDash_{[b,e]} \Diamond D \\
\Longrightarrow & \text{By definition of } \Diamond \\
& I \vDash_{[b,e]} tt \frown D \frown tt \\
\Longrightarrow & \text{Theorem 8.2.4, 8.2.7} \\
& I_s \vDash_{[s(b),s(e)]} tt \frown D \frown tt \\
\Longrightarrow & \text{By definition of } \Diamond \\
& I_s \vDash_{[s(b),s(e)]} \Diamond D
\end{array}$$

□

A similar proof can be used to show that if a duration formula D is interval-compress truth preserving $\mathit{isutp}(D)$, then $\Diamond D$ is also interval-compress truth preserving $\mathit{isutp}(\Diamond D)$.

Using the *eventually* operator and a pair of negations, the *always* operator is constructed. This is a very powerful operator because the satisfaction of *always* D ($\Box D$) means that the formula D holds on all subintervals. Recall that the negation changes an interval-stretch truth preserving formula into an interval-compress truth preserving formula. In this case however, since there are two negations, the effect of each negation cancels the other's.

Theorem 8.2.9. If a duration formula D is interval-stretch truth preserving *isdtp*(D), then $\Box D$ is also interval-stretch truth preserving *isdtp*$(\Box D)$:

Proof.

$$\begin{array}{ll} & \mathit{isdtp}(D) \\ \Longrightarrow & \text{Theorem 8.2.3} \\ & \mathit{isutp}(\neg\, D) \\ \Longrightarrow & \text{Theorem 8.2.8} \\ & \mathit{isutp}(\Diamond\neg\, D) \\ \Longrightarrow & \text{Theorem 8.2.3} \\ & \mathit{isdtp}(\neg\, \Diamond\neg\, D) \\ \Longrightarrow & \text{By definition of } \Box \\ & \mathit{isdtp}(\Box D) \end{array}$$

□

A similar proof can be used to show that if a duration formula D is interval-compress truth preserving *isutp*(D), then $\Box D$ is also interval-compress truth preserving *isutp*$(\Box D)$.

The following theorem considers the conjunction of two duration formulae. This operator also preserves truth if the sub-formulae do so.

Theorem 8.2.10. If duration formulae D and E are interval-stretch truth preserving *isdtp*(D) and *isdtp*(E), then their conjunction is also interval-stretch truth preserving *isdtp*$(D \wedge E)$:

$$I \vDash_{[b,e]} D \wedge E \Rightarrow I_s \vDash_{[s(b),s(e)]} D \wedge E$$

Proof.

$$\begin{array}{ll} & I \vDash_{[b,e]} D \wedge E \\ \Longrightarrow & \text{By definition of validity of conjunctions} \\ & I \vDash_{[b,e]} D \text{ and } I \vDash_{[b,e]} E \\ \Longrightarrow & \mathit{isdtp}(D) \text{ and } \mathit{isdtp}(E) \\ & I_s \vDash_{[s(b),s(e)]} D \text{ and } I_s \vDash_{[s(b),s(e)]} E \\ \Longrightarrow & \text{By definition of validity of conjunctions} \\ & I_s \vDash_{[s(b),s(e)]} D \wedge E \end{array}$$

□

Similarly, it can be shown that if duration formulae D and E are interval-compress truth preserving *isutp*(D) and *isutp*(E), then their conjunction is also interval-compress truth preserving *isutp*$(D \wedge E)$.

Finally, we consider the disjunction of two duration calculus formulae and show that this operator also preserves truth.

Theorem 8.2.11. If duration formulae D and E are interval-stretch truth preserving *isdtp*(D) and *isdtp*(E), then their disjunction is also interval-stretch truth preserving *isdtp*$(D \vee E)$:

$$I \vDash_{[b,e]} D \vee E \Rightarrow I_s \vDash_{[s(b),s(e)]} D \vee E$$

Proof.

$$\begin{array}{ll} & I \vDash_{[b,e]} D \vee E \\ \Longrightarrow & \text{By definition of validity of disjunctions} \\ & I \vDash_{[b,e]} D \text{ or } I \vDash_{[b,e]} E \\ \Longrightarrow & \mathit{isdtp}(D) \text{ and } \mathit{isdtp}(E) \\ & I_s \vDash_{[s(b),s(e)]} D \text{ or } I_s \vDash_{[s(b),s(e)]} E \\ \Longrightarrow & \text{By definition of validity of disjunctions} \\ & I_s \vDash_{[s(b),s(e)]} D \vee E \end{array}$$

□

Similarly, it can be shown that if duration formulae D and E are interval-compress truth preserving *isutp*(D) and *isutp*(E), then their disjunction is also interval-compress truth preserving *isutp*$(D \vee E)$.

8.2.3 Events on Boolean Variables

In our case studies (see Chapter 9), particularly the second one, we show the usefulness of events in practical situations. The advantage of considering events is that they are not affected by slowing down and speeding up because they take zero time. For the definition of events we will use the definition as adopted for counterexample traces (see Subsection 3.3.1).

Before proving that $\updownarrow P$ is interval-stretch invariant, we show that both $\nwarrow P$ and $\nearrow P$ are interval-stretch invariant. They are thus interval-compress invariant as well.

Proposition 8.2.8. $\nwarrow P$ and $\nearrow P$ are interval-stretch invariant.

Proof. First we start by showing that $\nwarrow P$ is interval-stretch truth preserving:

$$I \models_{[b,e]} \nwarrow P \Rightarrow I_s \models_{[s(b),s(e)]} \nwarrow P$$

$$\begin{aligned}
& I \models_{[b,e]} \nwarrow P \\
\Longrightarrow\; & \text{By definition of } \nwarrow P \\
& b = e \wedge \exists\, m : \mathbb{R} \;\cdot\; m < b \wedge I \models_{[m,b]} \lceil P \rceil \\
\Longrightarrow\; & \text{Applying function } s \\
& s(b) = s(e) \wedge \exists\, m : \mathbb{R} \;\cdot\; m < b \wedge I \models_{[m,b]} \lceil P \rceil \\
\Longrightarrow\; & \lceil P \rceil \text{ is interval-stretch truth preserving} \\
& s(b) = s(e) \wedge \exists\, m : \mathbb{R} \;\cdot\; s(m) < s(b) \wedge I \models_{[s(m),s(b)]} \lceil P \rceil \\
\Longrightarrow\; & \text{Substitution by } n = s(m) \\
& s(b) = s(e) \wedge \exists\, n : \mathbb{R} \;\cdot\; n < s(b) \wedge I \models_{[n,s(b)]} \lceil P \rceil \\
\Longrightarrow\; & \text{By definition of } \nwarrow P \\
& I_s \models_{[s(b),s(e)]} \nwarrow P
\end{aligned}$$

□

It is very similar to show that $\nwarrow P$ is also interval-compress truth preserving. By Proposition 8.2.7, we know that an interval-compress truth preserving formula preserves falsity on interval-stretches. Thus, $\nwarrow P$ is interval-stretch invariant. Using Corollary 8.2.2, $\nwarrow P$ is also interval-compress invariant.

The same reasoning can be used to prove that $\nearrow P$ is also interval-stretch invariant and interval-compress invariant.

Finally, we show that $\updownarrow P$ is also interval-stretch invariant and interval-compress invariant.

Theorem 8.2.12. The formula $\updownarrow P$ is interval-stretch invariant:

$$I \models_{[b,e]} \updownarrow P \Rightarrow I_s \models_{[s(b),s(e)]} \updownarrow P$$

Proof.

$$
\begin{array}{ll}
 & I \models_{[b,e]} \updownarrow P \\
\Longrightarrow & \text{By definition of } \updownarrow P \\
 & I \models_{[b,e]} (\nwarrow \neg P \wedge \nearrow P) \vee (\nwarrow P \wedge \nearrow \neg P) \\
\Longrightarrow & \nwarrow P \text{ and } \nearrow P \text{ (Proposition 8.2.8), conjunction (Theorem 8.2.10) and} \\
 & \text{disjunction (Theorem 8.2.11) are slowdown truth preserving} \\
 & I_s \models_{[s(b),s(e)]} \updownarrow P
\end{array}
$$

□

Using Corollary 8.2.2, $\updownarrow P$ is also interval-compress invariant.

This concludes the set of proofs showing which duration calculus fragments preserve truth on interval-stretches and which preserve truth on interval-compressions. In the next section, we will discuss practical applications of this theory.

8.3 Proposed Applications of SDTP/SUTP

Runtime verification may be used to monitor a third party system which was not created with runtime verification in mind. If we introduce monitoring on the system, we will slowdown its execution. This may result in violating properties which would hold if the monitoring was not introduced. Similarly, this may result in satisfying properties which would not have been satisfied if the system was not slowed down. If the properties used for such a system are SDTP properties then, no matter how much the system is slowed down, if a property used to hold in the normal running of the system, it will also hold in a slowed down version of the same run. However, if a SDTP property did not hold in the normal execution of the system, it may hold on the slowed down version. To avoid this, one must use a property which is *false preserving.* There is a subset of properties which are both slowdown truth preserving and slowdown false preserving: slowdown invariant. In this case the result (satisfaction/violation) of properties will remain intact when the system is slowed down.

Runtime verification may be used during testing and then removed from the system upon deployment to get rid of the overhead. In this case the target system will presumably work faster without monitoring than with monitoring. Ensuring that properties are SUTP, would provide the assurance that the properties which were verified during testing will still remain true when the system runs faster. As in the case of verifying third party systems, this does not guarantee that a property which was violated in a normal execution, cannot be satisfied in the faster version. In order to obtain this guarantee as well, speedup invariant properties should be used. One should note that the set of slowdown invariant and speedup invariant properties are in fact the same set of properties.

8.4 Conclusion

Considering the difficulty of monitoring real-time properties, we consider our approach of slowdown/speedup truth preservation as a good compromise which can be useful in practical situations. The subset of duration calculus which falls under slowdown (speedup) truth preservation is substantial. Furthermore, there is also a subset of properties about events which is slowdown (speedup) invariant. We can identify a lot of useful properties which simply ensure that *something* occurs after a particular number of events. We also feel that this area of research is still in its infancy. We hope that the idea is taken up and further developed to make the monitoring of real-time properties more practical.

Part IV

Experiments

9. Case Studies

9.1 Introduction

To show the practicality of a system it is essential to use it in real-life scenarios. In this section, we apply the LARVA architecture to two practical systems: a transaction processing system and a network monitoring system. The first case study was carried out before LARVA was completed and therefore, not all the features of LARVA were used. Some features of LARVA were, in fact, motivated by this case study. Furthermore, the transaction processing system is a real-life system which belongs to an industrial company with high security concerns. This fact enables us to comment on the place of runtime verification in an industrial software-developing company. The second case study is more focused on the applicability of real-time properties in real-life scenarios. The network monitoring system was developed with runtime verification in mind. Therefore, the challenges are more focused on the correct expression of properties, rather than the implementation and technical aspects of integrating the monitoring system with the target system.

In the following two sections we will go into the details of both case studies. The last section concludes the chapter.

9.2 A Transaction Processing System

During its development, LARVA was used on an real-life system which handles credit card transactions. The complexity of this system lies, not only in the size of the underlying code (which is over 26,000 lines of code), but also in the security implications and communication required among various components (including third party systems, such as banks). The system is designed to hold sensitive information of thousands of people, so the implications of a single leak of sensitive information could undermine the confidence of the users in the system, leading to drastic consequences.

The system is composed of two parts: one handling the transactions in a database and the other handles the communication to the entity which is involved in the transaction. These will be referred to as the *transaction handling system* and the *processor communication system*, respectively. The whole system will be referred to as the *transaction system*.

9.2.1 Verified Properties

A number of properties which were verified on the transaction processing system using Larva are described below.

Logging of Credit Card Numbers

During the development of the original transaction system, credit card numbers were logged for testing purposes. This is, however, not in line with standard practice of secure handling of credit card numbers. These logging instances were manually removed from the code. However, to ensure that no instances remained, a simple verification check is applied to ensure that no data resembling a credit card number is ever logged.

The logic involved can be easily represented by the Larva automaton shown in Figure 9.1.

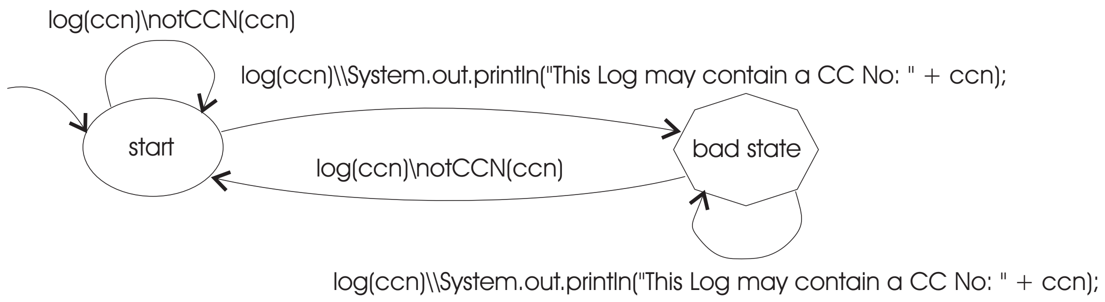

log(ccn)\notCCN(ccn)
log(ccn)\\System.out.println("This Log may contain a CC No: " + ccn);
start
bad state
log(ccn)\notCCN(ccn)
log(ccn)\\System.out.println("This Log may contain a CC No: " + ccn);

Figure 9.1: Combining all the components.

In this case, the transitions of the automaton are all triggered by the event *log* which represents any method call involving the *Logger*. A condition is required on each transition (after the first backslash) to distinguish those strings which potentially contain a credit card number from those which surely do not. To this end, a method *NotCCN* was implemented which, given a string, returns true if the string does not contain any credit card numbers. The automaton starts looping on the start state. Once a potential credit card number is found, the automaton moves to the *bad_state* and outputs a message to the console.

A snippet of the script which monitors this property is as follows:

```
GLOBAL
{
  EVENTS {
    log(String ccn) = {Logger l.*(ccn)}
  }
  PROPERTY logging
  {
    STATES {
       BAD { bad_state }
       STARTING { start }
    }

    TRANSITIONS
    {
      start -> start         [log(ccn)\NotCCN(ccn)]
      start -> bad_state     [log(ccn)\\
        System.out.println("This Log may contain a CC No: " + ccn);]
      bad_state -> start     [log(ccn)\NotCCN(ccn)]
      bad_state -> bad_state [log(ccn)\\
        System.out.println("This Log may contain a CC No: " + ccn);]
    }
  }
}
```

When this script is translated to Java with the LARVA compiler, two main files will be generated: an AspectJ file which "gathers" the relevant events from the system, and a class file which implements the automaton. The following shows the pointcut which the AspectJ file contains when the above script is translated:

```
before(String ccn, Logger l) : (call(* Logger.*(..)) && target(l)
  && args(ccn)
```

The following shows a part of the automaton implementation in the generated class file, which handles what happens when the system is in the *start* state.

```
else if (_state_id_logging == 1)
{
  if (1==0) {}
  else if ((_occurredEvent(_event,0/*log*/)) && (NotCCN (ccn)))
  {
    _state_id_logging = 1; // moving to state start
    _goto_logging(_info);
  }
  else if ((_occurredEvent(_event,0/*log*/)))
  {
    _cls_Logger0.pw.println("This Log may contain a CC No: " + ccn );
    _state_id_logging = 0; // moving to state bad_state
    _goto_logging(_info);
  }
}
```

The generated Java files are assumed to belong to a package named *larva*. Therefore, these should be put in a folder with this name and the system should be re-compiled. Once this is completed, the system will be monitored by the logic described in the script.

Output of the Verification System The output of the monitoring code is all redirected to a text file with the name “output_Logger.txt” (this includes any *System.out.println* in the script file and any automaton-generated text).

The following shows that a bad state has been reached when “Transaction 2008032011254001901 Passthrough Pair Stored” is logged. In this case the transaction id, incidentally, is a valid credit card number and the verification system detected it. Upon detection, the automaton reverts to a bad state and outputs the stack trace of the code which triggered the bad state. This is shown below:

```
AUTOMATON::> logging() STATE::>start
MOVED ON METHODCALL: Logger.debug(Object) TO STATE::> start

AUTOMATON::> logging() STATE::>start
MOVED ON METHODCALL: Logger.debug(Object) TO STATE::> start

AUTOMATON::> logging()
STATE::>start
  This Log may contain a CC No: Transaction
    2008032011254001901 Passthrough Pair Stored
MOVED ON METHODCALL: Logger.debug(Object) TO STATE::> bad_state
```

Transaction Execution

Transactions are processed by going through a number of stages, including authorisation, communication with the user interface, insertion of the transaction in the database and communication with the commercial entity involved in the transaction. The stages taken depend on the type of transaction. It is quite straightforward to design a property to ensure that a transaction goes through the proper stages. This is especially true since the states of our automata-based language can be used to model the stages of the transaction. Other properties regarding the life cycle of a transaction are defined as explained below.

Events We first require an event to detect the start of a transaction execution. This is the method call *processTransaction*. Then we need to detect when the state id is being set and when each state is being processed (*setCurrentStateID* and *State.process* respectively). Finally, we need to detect when the *processTransaction* method completes, to check whether the transaction was approved or not. The complete list of events is shown below:

```
proc(RunnableObject ro)      = {ro.processTransaction()}
    where {t=ro.getTransaction();}

setid(Transaction t1,int sid)= {t1.setCurrentStateID(sid)}
    where {t=t1;}

procState(RunnableObject ro) = {State.process(tsm)}
    where {t=tsm.getTransaction();}
```

```
afterproc(RunnableObject ro) = {{ro.processTransaction()
uponReturning()}|{ro.processTransaction() uponThrowing()}}
    where {t=ro.getTransaction();}

all() = {proc | setchain | setid | procState | afterproc}
```

Worth noticing in this list are the two events which are collections: *afterproc* and *all*. For example, *afterproc* includes the two ways of completing the method *processTransaction*: either through a normal return or through an exception throw.

Comparing Transactions To implement this property we need to be able to distinguish among the transactions which are being currently executed. Therefore, we need some way to compare transactions and decides on their equality. This is not as trivial as it may seem because the transaction object does not have an *equals* method. Comparing the pointer value of the object is also useless because the pointer changes during the communication of the transaction. The reason is that serialisation (used for communication) reconstructs a new instance of the transaction. Therefore, a transaction's uniqueness should be based on its id. However, this is not available at the time of the transaction construction. Thus, before an id is actually assigned to the transaction, we need some other way to decide equality.

This issue was resolved by allowing the user to define a custom method which compares two transaction objects and decide whether they are the same object or not. The custom *equals* method implemented (named *equalT*), first uses pointer equality and then, when the generated id is available, the id is used instead. Furthermore, we also need a way to represent the transaction for the display purposes of the automaton. Hence, we also allow the user to specify a method which turns a transaction into a string (named *toStringT*). To express the fact that we need an automaton for each transaction the user must enclose the declaration of events, states and transitions within a *FOREACH* section. The declaration is as follows:

```
FOREACH (Transaction t equateUsing equateT stringUsing toStringT)
```

Transitions After the events have been chosen, and the context is settled, the next step is to formulate the actual automaton. In this example, we have to check that the current state of the transaction is the right one. Each transaction has to go through a list of states. If one of these states is repeated or skipped, the automaton should report a bad state. The checking starts upon the call of *processTransaction*. At this point, the variable *state* is set to the transaction's current *state id*. Eventually, when the method *setCurrentStateID* is called, it is ensured that the id being set is one greater that the current id (stored in our local variable). The transaction is considered ready when the state id reaches the number of states in the list. Finally, the automaton reaches the *complete* state when the transaction is approved.

To give an idea of how the above logic would be encoded, we give the following snippet of transitions:

```
TRANSITIONS
{
  start -> process_trans [proc\\if (state==-1)
    state=ro.getTransaction().getCurrentStateID();]

  process_trans   -> set_state [setid\sid == state + 1\state = sid;]

  set_state -> process [procState\tsm.getTransaction().equals(t) &&
    t.getCurrentStateID()==state]

  set_state -> bad_state [all()]

  ...
}
```

Detecting a Violation To simulate a possible violation of this property we altered the code in the processor communication system. The alteration incremented the current state id of the transaction. When the transaction is communicated back to the transaction handling system, the violation is detected as shown below:

```
AUTOMATON::> generaltransaction(2008032016135012701)
STATE::>process MOVED ON METHODCALL:
Transaction.setCurrentStateID(short) TO STATE::> set_state

AUTOMATON::> generaltransaction(2008032016135012701)
STATE::>set_state MOVED ON METHODCALL:
State.process(TSMRunnableObject) TO STATE::> process

AUTOMATON::> generaltransaction(2008032016135012701)
STATE::>process MOVED ON METHODCALL:
Transaction.setCurrentStateID(short) TO STATE::> bad_state
```

One may argue that the possibility of such a violation occurring is remote. However, since the transaction is being communicated to another entity, the external code may be maliciously altered. Furthermore, it may still be useful to keep “extra” checks during testing. In this way, any unintentional alteration of code, which causes a violation of the verified properties, will be detected.

Authorisation transactions

Authorisation transactions have to be checked to ensure that all the stages are processed in the correct order, keeping certain values unchanged — for instance, one must ensure that the transaction amount is not changed after being set.

An aspect of this property which was not used so far in the case study is the use of invariant. These allow the user to specify the relationship between a transaction before it goes through a transition and the transaction after it goes through it. In our case, we wanted to check that the amount of the transaction remains unchanged. Thus, the invariant is applied on the method *getAmount* as shown in the following code:

```
INVARIANTS {
  Double amountInvariant = t.getAmount();
}
```

The check on the amount is useless unless the amount has been set. For this reason the invariant is enabled after a particular transition as shown below:

```
merchant_loader --> trans_inserter [transIns]
  [enable amountInvariant]
```

This means that after the call to process the *transactionInserter* state, the *getAmount* method will be called whenever the automaton moves to a new state and its return value will be compared to the transaction amount as it was at the last transition.

Detecting a Violation The verification mechanism was tested by altering the amount of a transaction to “123456” at the system of a third party. As soon as the transaction returns through communication, the violation is detected and the output of the automaton is as shown:

```
Amount being compared: 123456
 Invariant Check Failed
```

Backlog

A particular feature of the system under scrutiny, is that if communication with a third party fails, the request is retried a number of times. After each communication failure, a delay is allowed before the next attempt. This process is called *backlogging*. The backlogs in the transaction handling system introduces new properties. For example, we must ensure that the backlog process is performed for the expected number of times or until the transaction is approved. A limitation is real-time in LARVA was not yet implemented at the time of the case study. If we had this possibility, we could use timeout mechanisms to ensure that retries occur within a given interval of time.[1]

To test the violation detection mechanism, we altered the transaction handling system to decrement the number of retries by two instead of one each time a retry is performed. This is detected as shown below when the number of expected remaining retries is 3 while the actual number in the transaction is 2.

[1]We had access to the proprietary code only for a limited period of time. Therefore, we were not able to try the latest version of LARVA on this case study.

```
AUTOMATON::> backlog(2008032414073001301 )
STATE::>BACKLOGCHAIN MOVED ON METHODCALL:
Transaction.setCurrentChainID(ChainID)
TO STATE::> set_chain

AUTOMATON::> backlog(2008032414073001301 )
STATE::>set_chain
  Mismatch found...Expected: 3 but found: 2
MOVED ON METHODCALL:
State.process(TSMRunnableObject) TO STATE::> bad_state
```

9.2.2 Evaluation

Considerations

Considering that the transaction system is a very big system, which is the product of weeks of work by a considerable number of developers, it was not easy to even get used setting up the necessary environment for it to run. Therefore, the evaluation should be taken into the context of the steep learning curve to get used to running and testing the transaction system. The time was also a notable issue especially when considering the time needed to simply compile and start a single test. Furthermore, there were various memory issues with the Java virtual machine when the AspectJ compiler was used with other technologies.

Another crucial consideration is that the experiment was carried out on a ready made system. As will be later discussed in the recommendations, it is suggested that the runtime verification progresses hand-in-hand with the software development life cycle.

While we were conducting the case study, we had to repeatedly ask information from the developers and testers of the transaction system. This was necessary because the code is not very well commented since the transaction system was initially meant to be a prototype not a complete system. Probably, the newer version would have been easier to work with because it is work-in-progress and developers would be better able to answer questions. Furthermore, the logic in the transaction system is mainly at the database level using stored procedures, while in new version the logic is in Java. This would have been a great advantage because Java is the language which we can verify with Larva. Although we did not uncover any unknown errors in the transaction system, we found all the errors which were intentionally placed in the code when we ran the appropriate tests.

Impact of the Case Study on Larva

Besides the intended objective of using a runtime verification tool for testing a real example, the case study served the purpose of testing Larva. In this sense, we got feedback showing the limitations of our tool. The current incarnation of Larva contains various features not in the original version, and which were identified to be necessary through the case study.

1. Larva was extended to handle two extra types of events: one which matches an exception throw and another which matches the handling of an exception (i.e. a catch block in Java), which were necessary for some properties.

2. The need for invariants was identified — a recurring property in a real-life scenario is to check that certain attributes are not changed when they are not supposed to change. Therefore, we extended our language to allow the user to specify such invariants and enabling/disabling them as necessary.

3. When handling invariants and context-sensitive properties Larva depends heavily on being able to calculate equality of objects — such a method may not necessarily be available for all objects, or a custom equality may be necessary to use. This issue was solved by allowing a user to specify a custom *equals* method.

4. No timeouts nor other real-time properties were supported. We have recently extended Larva to include clocks and stopwatches.

Appraisal of the Case-Study

We feel that the verification of the transaction system was an excellent academic exercise because we were in a good position to understand what language expressivity is required to verify real-life systems. Moreover, using our compiler to translate Larva properties into Java code, has acted as a good testing ground for our compiler, making it better as problems were uncovered and resolved.

Although, more interesting properties might have been checked if there was more time available, overall, the case-study has demonstrated the benefits of using this type of runtime verification. After all, our main aim was a proof of concept on an old version of the transaction system. One might still argue that the verified properties simply verify basic programming logic. However, checking the programming logic has various benefits:

- First, writing the properties helps the writer to understand the system specification better.

- Secondly, once a verification script is written, any future alteration of code can automatically be tested to still adhere to the system specification. This increases the reliability of both the current and the future code.

- Thirdly, (and from the transaction system experience, most importantly) the verification can act as an aid for testing. This can help as an easier alternative to manual line-by-line debugging by reporting the necessary output from the appropriate method calls.

Measurements

Given the nature of the system, with different components communicating and synchronising their behaviour, it was difficult to measure the overhead of the monitoring system for the case study. This, and the lack of time are the reasons why we do not have any concrete measurements of the resources used by the system under consideration.

9.2.3 Recommendations

From the experience we had in this case study, we will give our opinion as to why runtime verification with Larva is beneficial for software development. Then, we will further discuss where in the development life cycle runtime verification should take place.

Motivations for Employing Runtime Verification with Larva

Sound software specification is the initial prerequisite for a successful software life cycle. We believe that an important aspect of runtime verification is that it forces the system architect/developer to reflect on the system properties so that they are explicitly declared. This in itself is a very healthy exercise which is very important for any software system. Once the system properties are clearly stated in an appropriate formal notation, then runtime verification comes almost for free. The only required step would be to relate the events of the system (used in the specification of properties) to concrete method calls in the implementation. The system can then be automatically verified and checked that it abides to the properties in the specification. Therefore, for such a little overhead we feel that the advantage of runtime verification is substantial.

Placing Runtime Verification in the Development Life Cycle

If we agree to introduce runtime verification in the development life cycle, the next question would be: where? In the previous section we have suggested that the specification of system properties should start from the beginning of the development life cycle. However, with no implementation yet available, we cannot relate the properties to the concrete events of the system. Therefore, runtime verification has to evolve alongside the implementation of the system.

We also observe an interesting relationship between runtime verification and testing. The reason is that without an actual execution of the system (be it a simulation as in testing or a real deployment) runtime verification cannot be performed. In fact, runtime verification can greatly help testers because it automatically deduces whether the test was successful or not. In some companies it is now a custom to start designing and implementing tests before the implementation of the system. This complements our idea that the system properties (for which we test) should be specified at the start of the software development life cycle. Eventually, when the implementation is ready for the tests to run, the runtime verification mechanism would be in place, automatically verifying whether the test was successful or not.

Who should come up with the System Properties?

This discussion leads to the question of who should come up with the system properties. The system architect? The developer? The tester? From our experience, it is first and foremost the task of the architect to draw up a set of system properties. This is backed up by the fact that for the system properties to be as complete as possible, their author should have a very intimate knowledge of the system as a whole. However, as already pointed out, the architect will have to base the properties on abstract events of the system, because the system has not yet been created. Therefore, it is then up to the developer to put the final connection between the relevant properties and his concrete part of the code. This ensures that the code being created adheres to the system properties. Furthermore, the tester should have a good enough knowledge of the properties so that he understands what the system is being verified for. Yet, this is implicit in the tester's role in the development life cycle.

If runtime verification is integrated to the development life cycle as proposed above, it should be seamlessly integrated with little effort. Moreover, keeping the system properties as the main focus of the life cycle (from the architecture to the testing) will undoubtedly help the stakeholders remain attentive to adhere to the specification.

A Two-Level Approach to System Properties

During the above discussion, we admitted that when the system is being specified, the properties have to be based on abstract events, which have not been yet implemented. This leads us propose a two-level approach to the system properties. Consider the property which checks that there should not be more than three bad logins in an hour. There are an infinite ways of implementing it in the system. For example if the developer blocks the user for an hour after two bad logins, the property will be satisfied. However, one can note that there may be quite a gap between the initial property stated at the system design stage and its implementation. Therefore, we propose that whilst the developer is implementing the system, he can formulate other properties which must be in line with the more abstract system properties. We argue that the more explicit the system properties are, the more clear it is how the system should behave and subsequently, the easier it is to monitor and verify that behaviour.

How Intrusive should the Verification Mechanism be?

An important issue is how intrusive should we allow the verifying mechanism to be. There is no simple answer to this dilemma. The work presented in the two previous chapters has the aim to give guarantees on the effect of introducing monitoring. These include guarantees on the memory and temporal requirements, and the effect of slowing down/speeding up a system on the satisfaction/violation of real-time properties. Another possible approach to reduce the impact of verification is to do it on a separate machine which is not necessarily in sync with the monitored system. This will be further investigated in future work.

9.2.4 Conclusion

Using Larva as a runtime verification language has so far proved to help the understanding and the explicit declaration of the system specification. The advantage of verification is that the implementation is automatically verified against the specification, bridging the gap between the two. This helps creating a top view of the system where the verification code is written in a centralized location without altering the system code. Using this approach we have caught errors which we have placed in the system ourselves. Given more time and better understanding of the system specifications, we might have also uncovered unknown errors. Nonetheless it is still useful to keep the verification mechanism in place so that whenever the code is altered no unintended errors are propagated.

This experience, has helped us to understand the practical side of runtime verification. Upon reflecting on this experience, we suggest a feasible way to integrate runtime verification with the software development life cycle. In a few words, we recommend that the system properties should be given prominence from the start of the life cycle by being explicitly declared. Having these declarations in place is the first and most important part of the runtime verification process. At the same time, declaring the system properties is part of any serious software development life cycle. With the little extra effort of specifying the properties formally in the Larva language, the system will be automatically monitored and verified to adhere to its specification.

We hope that this case-study will contribute towards better understanding and appreciation of runtime verification. We hope even more that the principles and techniques described will contribute to better software especially with respect to reliability and robustness.

9.3 Network Monitoring System

Effective network monitoring is a priority for any entity which relies on a network. Automatic intrusion detection systems are popular but these may not allow the user to program custom security properties and extend the detection system. Using the Larva architecture it is relatively easy to provide such an extensible and programmable monitoring system. Creating and testing a network monitoring system is a very valuable to test the expressivity and applicability of the Larva architecture.

A simple packet sniffer was created using Jpcap technology to capture packets being sent and received on a particular network interface. Depending on the sequence of packets, events are generated to be monitored by the system. In total, five different properties where constructed and tested: four using counterexample traces (translating them into Larva) and another using Larva directly. Three of the four were also implemented using QDDC or Lustre and then translated into Larva. Recall that a fragment of QDDC can be translated into Lustre, and Lustre can be translated into Larva. QDDC is quite limited and in the examples below it is usually more convenient to write the properties directly in Lustre. The reasons will be highlighted later on.

All of the properties featured in this case-study have been tested and the attacks were detected as expected. More details about each property will be given in the following subsections.

9.3.1 Properties in Counterexample Traces and QDDC/Lustre

Truth Preservation and Memory Upperbound

For the properties which have been written in counterexample traces and involve real-time, we applied the theory we developed to identify which properties are slowdown truth preserving or speedup truth preserving. In this case study, since the aim is to limit the frequency of certain types of packets, all of the properties are slowdown truth preserving: i.e. if the property holds on a particular run of the system it will also hold on a slowed-down version of that run. A practical implication is that monitoring the system (and possibly slowing it down) will not violate any of the properties which used to hold on the faster sequence of packets. Note that the first property, which does not involve real-time, is both slowdown and speedup truth preserving.

For some of the properties, we also provide the equivalent logic in QDDC/Lustre. This is not possible for all the properties because both QDDC and Lustre have a number of limitations. The limitation will be explained for each property in the following subsections. The advantage of writing properties in QDDC/Lustre is that they can be monitored with an upperbound memory known at the time of compilation. However, this guarantee only applies for a single automaton. For the properties where the automata are replicated for each context, the guarantee still holds for each automaton but the memory required grows with the number of unique contexts.

Initiating Connections

For strict security concerns, one may wish to disable any incoming TCP packets which do not belong to connections initiated by the host machine being monitored. The problem with this property is that if the packet sniffer is started while a connection is ongoing, the property will be violated when in fact the connection might have been initialised from the host machine earlier on. The initialisation of a TCP connection requires a complete three-way handshake: first a synchronization packet from the client, then a synchronization and acknowledgement packet from the server and another acknowledgement from the client. If the host machine receives a synchronization packet without having sent one beforehand, then an outsider is trying to open a connection. This property may be written as a counterexample trace:

$$\neg\,(\boxminus\, sendSYN \frown \updownarrow receiveSYN \frown true)$$

Since a TCP connection is a four tuple (an address and port for the server and the client), this property has to be monitored for each distinct tuple. This means that if the host machine is connected on a particular port with a server, the server cannot initiate a connection through another port to the host machine.

Details The counterexample formula was first translated into a Larva automaton. Subsequently, the events required to trigger the automaton were defined. Each of these events is also responsible of changing the value of the state variables. State variables are required by the automaton to choose a destination state from the current state. In this example, both *sendSYN* and *receiveSYN* are initially false. If *sendSYN* turns to true, then the property cannot be satisfied for the connection being considered. In a counterexample formula there is no notion of context which in this case is the four-tuple connection. Thus, another required modification is to add context to the Larva script (the output of the translation from the counterexample formula). With these modifications the network traffic can be monitored for incoming TCP connection requests. A snippet of the modified Larva script is shown below (Note the four-tuple context and the events assigning a value to the state variables *rSYN* and *sSYN* representing received and sent (respectively) synchronisation packets.):

```
FOREACH(InetAddress ip, InetAddress ip2, Integer port1, Integer port2) {

  EVENTS {

    receiveSYN(boolean rSYN, boolean sSYN)
     = {*.receiveSYN(InetAddress src, InetAddress dst, int src_port, int dst_port)}
 where {rSYN = true; sSYN = false; ip = src; ip2 = dst; port1 = src_port;
        port2 = dst_port;}
    sendSYN(boolean rSYN, boolean sSYN)
     = {*.sendSYN(InetAddress src, InetAddress dst, int src_port, int dst_port)}
 where {rSYN = false; sSYN = true; ip = src; ip2 = dst; port1 = src_port;
        port2 = dst_port;}
    all() = {receiveSYN | sendSYN}
}
```

QDDC/Lustre Monitoring for incoming connections is quite simple and this can be easily represented in QDDC:

$$\lceil\lceil \neg\, sendSYN \wedge \neg\, receiveSYN \rceil\rceil \; then \; begin(receiveSYN)$$

Note that this formula must also be monitored for each connection context. Therefore, after the translation into Larva, the necessary modifications should be carried out to provide the context and relate the variables with the system events.

Redirect Messages

In the case of a machine with a routing table, a lot of ICMP redirect messages can cause the system to slow down. Therefore, if a lot of ICMP redirect messages are received in a relatively short time interval, this may be considered as a threat to the system. The property can be written as follows:

$$\begin{aligned} \neg\, (&true \frown \; \updownarrow redirectMsg \frown \lceil true \rceil \wedge \ell < 2 \\ &\frown \; \updownarrow redirectMsg \frown \lceil true \rceil \wedge \ell < 2 \frown \; \updownarrow redirectMsg \frown true) \end{aligned}$$

Note that $\lceil true \rceil$ between the events is not redundant because this ensures that the length of the interval is greater than zero (by definition of $\lceil - \rceil$). Allowing an interval with zero length between two events would mean that the events are in fact the same event. With little modification, the property can be used to monitor against repeated ping messages and other unwanted (possibly malicious) traffic. In the case of the redirect packets, we do not feel the need to monitor the same property for each individual ip address. If the redirect messages are arriving repeatedly without sufficient time interval between them (no matter what the source ip address is), then a property violation is signalled so that the user can take the appropriate action.

Details The compilation process of this property into the monitoring code is very straight forward since there is no need of a context. The only modification required is to relate the appropriate system events with the variables monitored.

QDDC/Lustre For this property it is difficult to construct the equivalent QDDC formula. This is because only the deterministic chop is available in QDDC fragment being considered. Thus, a property which should be checked for each subinterval cannot be represented in QDDC. In such a case, it is easier to use the Lustre language directly. The code would be as follows:

```
node redirectMSG (_rt_clock:int; redirect:bool) returns (violated:bool);
var
 cnt  :int;
 time :int;
```

```
let
  time = if (false -> pre redirect) then (0 -> pre _rt_clock)
         else if ((0 -> pre cnt) == 0) then _rt_clock
         else (0 -> pre time);
  cnt = if (redirect and ((_rt_clock-time) < 2000)) then ((0 -> pre(cnt)) + 1)
        else if (_rt_clock - time < 2000) then (0 -> pre cnt)
        else if (redirect) then 1
        else 0;
  violated = if (cnt >= 3) then true else (false -> pre violated);
tel
```

Connection Failure Retries

A possible denial of service attack is to initiate an excessive number of connection initialisations to a server and then leaving the handshake incomplete. The server will have to wait for each of these initialisations to timeout. Sometimes these timeouts can cause serious availability problems for the server because connection requests can be issued at very high speeds. A simple check would be to limit the number of subsequent failed connection retries originating from the same ip address. (In Section 9.3.2 we explain how a more advanced property can be used against more sophisticated denial of service attacks) The property that prohibits three subsequent failed connection attempts, which occur in a short period of time, may be written as follows:

$$\begin{aligned}\neg\,(&\mathit{true} \frown\ \updownarrow \mathit{failedConn}(\mathit{ip}) \frown \lceil \mathit{true} \rceil \wedge \ell < 1 \wedge \boxminus\ \mathit{successConn}(\mathit{ip})\\ &\frown\ \updownarrow \mathit{failedConn}(\mathit{ip}) \frown \lceil \mathit{true} \rceil \wedge \ell < 1 \wedge \boxminus\ \mathit{successConn}(\mathit{ip})\\ &\frown\ \updownarrow \mathit{failedConn}(\mathit{ip}) \frown \mathit{true})\end{aligned}$$

To monitor the above property, we can use two sub-properties which define what it means for a connection to be successful or to fail. A successful connection should always violate the following counterexample formula:

$$\begin{aligned}\neg\,(&\mathit{true} \frown\ \updownarrow \mathit{SYN}(\mathit{ip}) \frown \lceil \mathit{true} \rceil \frown\ \updownarrow \mathit{serverSYNACK} \frown \lceil \mathit{true} \rceil \wedge \ell < 5\\ &\frown\ \updownarrow \mathit{ACK}(\mathit{ip}))\end{aligned}$$

Similarly, a TCP handshake which is not acknowledged within 5 seconds should always violate this formula:

$$\begin{aligned}\neg\,(&\mathit{true} \frown\ \updownarrow \mathit{SYN}(\mathit{ip}) \frown \lceil \mathit{true} \rceil \frown\ \updownarrow \mathit{serverSYNACK}\\ &\frown \ell > 5 \wedge \boxminus\ \mathit{ACK}(\mathit{ip}))\end{aligned}$$

Using these two sub-formulas we can more clearly monitor successive connection failures from a particular ip address. For this purpose Larva channels can be used to connect properties together as explained in the following details.

Details The compilation of this property is more complex than the previous example because three properties must communicate together to form one more complex property. Furthermore, the sub-properties monitoring the successful or failed handshake must be "restartable" for each context. This means that as soon as the monitoring automaton reaches a particular state, it must be discarded so that later, the same automaton can start again from the starting state. In our example, as soon as a valid handshake is found, the automaton restarts so that another connection with the same context can be monitored.

Apart from these features, this property is particularly interesting because the different properties require different levels of context. A TCP handshake must be monitored at a connection level with the usual four parameters (ip addresses and port numbers) while the monitoring of successive failed connection retries is performed for each individual ip address.

The compilation of this property was carried out as follows: first the three sub-properties were separately translated into Larva automata and then placed into one Larva script file. Two channels were declared: *successChannel* and *failureChannel*. The property monitoring successful handshakes sends an event on the *successChannel* while the property monitoring failed handshakes sends an event on the *failureChannel*. The channels both transmit the events to the property which monitors the frequency of failed handshakes. Note that the events sent on the channel contain the ip address which initiated the handshake. This is important because the frequency of connection failures is monitored for each ip address.

QDDC/Lustre As with the case of redirect messages, QDDC is not useful to monitor such a property. Furthermore, it is not possible to represent a failed handshake in Lustre because this requires a check upon a timeout. In Lustre, time events cannot be generated. Thus, for this property we do not have the equivalent property in QDDC/Lustre.

Port Scan

A port scan is the discovery of open ports on a system by trying to initialise connections on a range of (usually) well-known ports. To monitor a port scan one should monitor frequent unsuccessful connections from an ip address to different ports. This can be written as follows (*port1, port2, port3* signify different ports):

$$\begin{aligned}\neg\,(&\mathit{true} \frown\ \updownarrow \mathit{connectTry}(\mathit{ip}, \mathit{port}1) \frown \lceil \mathit{true} \rceil \wedge \ell < 2\\ &\frown\ \updownarrow \mathit{connectTry}(\mathit{ip}, \mathit{port}2) \frown \lceil \mathit{true} \rceil \wedge \ell < 2\\ &\frown\ \updownarrow \mathit{connectTry}(\mathit{ip}, \mathit{port}3))\end{aligned}$$

Details Although this property seems very similar to the property concerning redirect messages, there is a significant difference because the port in each connection try must be different from the others. This prerequisite cannot be expressed as a counterexample formula. Thus apart from translating the above formula into a Larva automaton, another automaton must be used to receive the raw events, check their port numbers, and forward the event only if the port is different from the last three ports numbers available. For this property to make sense, it has to be monitored for each ip address. Therefore, another modification is required in the Larva script to take this into consideration. A code snippet showing the changes is shown below (Note the channel *distinctPort* required for the communication between the two automata.):

```
FOREACH (InetAddress ip)
{
  VARIABLES {
    Clock c2;
    Clock c3;
    int port1 = -1;
    int port2 = -1;
    int port3 = -1;
    Channel distinctPort;
  }

  EVENTS {
    receive(port)           = {*.packetReceived(InetAddress ip1, int port)}
                               where {ip = ip1;}
    distinct(boolean rPckt) = {distinctPort.receive(Object ip2)}
                               where { rPckt = true; ip = (InetAddress)ip2;}
    c3AT0_01(boolean rPckt) = {c3@5} where {rPckt = false;}
    c2AT0_01(boolean rPckt) = {c2@5} where {rPckt = false;}
    all()                   = {distinct | c3AT0_01 | c2AT0_01}
  }

  PROPERTY ports {
    STATES {
      NORMAL { normal1 normal2 }
      STARTING { start }
    }

    TRANSITIONS {
      start   -> normal1 [receive\port1 != port && port2 != port
        && port3 != port\port1 = port;distinctPort.send(ip);]
      normal1 -> normal2 [receive\port1 != port && port2 != port
        && port3 != port\port2 = port;distinctPort.send(ip);]
      normal2 -> start   [receive\port1 != port && port2 != port
        && port3 != port\port3 = port;distinctPort.send(ip);]
    }
  }
}
```

QDDC/Lustre In Lustre, the modularity is inherent in the nodes. Thus, the two automata required in Larva are reflected by two nodes in Lustre. The following Lustre code shows a node which upon receiving a port number, returns true if the port number is different from the last three port numbers received. The second node returns true if three packets with unique ports are received with less than 2000 milliseconds between each two subsequent packets.

```
node uniqueport (receive: bool; port: int) returns (unique:bool);
var
port1,port2,port3 :int;
cnt :int;
let
  unique = if (receive and port != pre port1 and port != pre port2
                and port != pre port3) then true else false;
  port1 = if (unique and pre cnt==0) then port
          else (-1 -> pre port1);
  port2 = if (unique and pre cnt==1) then port
          else (-1 -> pre port2);
  port3 = if (unique and pre cnt==2) then port
          else (-1 -> pre port3);
  cnt = if (unique) then ((0 -> pre cnt)+1)%3 else (0 -> pre cnt);
tel

node portscan (_rt_clock:int; unique:bool) returns (violated:bool);
var
 cnt  :int;
 time :int;
let
  time = if (false -> pre unique) then (0 -> pre _rt_clock)
         else if ((0 -> pre cnt) == 0) then _rt_clock
         else (0 -> pre time);
  cnt = if (unique and ((_rt_clock-time) < 2000)) then ((0 -> pre(cnt)) + 1)
        else if (_rt_clock - time < 2000) then (0 -> pre cnt)
        else if (unique) then 1
        else 0;
  violated = if (cnt >= 3) then true else (false -> pre violated);
tel

node main (_rt_clock:int; receive: bool; port: int) returns (violated:bool);
let
  violated = portscan (_rt_clock, uniqueport (receive, port));
tel
```

9.3.2 More Complex Properties Written in Larva

A common denial of service attack is to initiate an excessive number of connection initialisations.A well crafted attack can be distributed among thousands of machines, causing a high volume of traffic to the target server. It is not easy to detect such an attack. A possible approach is to use statistics. These can be used to elicit patterns of normal usage. Useful statistics include the number of pending connection requests, the number of open connections, the number of distinct ip addresses currently connected, and so on. These can be used to detect abnormal use and possibly identify all malicious users. The approach we chose is relatively simply and may not be able to identify malicious users, although as explained below it is able to detect abnormal use if an attack is carried out using TCP synchronisation packets.

Details

The main purpose of this property is to be able to detect that a host is under attack and possibly identify ip addresses which are conducting an attack on the monitored host. The approach to achieve this goal is to use the relationship between the number of pending connections and the number of open connections. If the number of pending connections remains larger than the number of open connections for a long period of time, this means that connections are not succeeding. Connection failures may be purposely used for a denial of service attack. Therefore, for each ip address, the number of pending connections can be compared to the average number of pending connections. If the number of pending connections remains higher than the number of open connections for a sufficient period of time, then the ip addresses whose number of pending connections exceed the average can be blocked. Blocking an ip address is relatively easy by using IPSec to modify the security policy being applied. This is done by executing operating system calls from Java.

This property is made out of three automata: (i) an automaton which monitors a TCP handshake, detecting a successfully open connection or a pending connection, and updating the corresponding count; (ii) an automaton which regularly monitors and compares the number of pending connections against the number of open connections; and (iii) an automaton which upon the detection of an attack, blocks the ip address which it monitors, if the number of pending connections is greater than the average pending connections.

Note that a copy of the first automaton is required for each TCP connection. Similarly, a copy of the third automaton is required for each ip address trying to initiate a connection. An attack is detected in the following number of steps: (i) copies of the first automata are continually monitoring connections and updating counts of pending and open connections; (ii) the second (global) automaton is repeatedly comparing the number of open and pending connections maintained by the connection automata; (iii) if the number of pending connections is found to be sufficiently high for a repeated number of times, an event is sent to the copies of the third automaton; (iv) as soon as automata of the third type receive an event, they check whether the number of pending connections is higher than the number of open ones for the individual ip address they monitor; finally (v) if a particular ip address has a number of pending connections which is higher than the average, it is block by its corresponding automaton (of the third type).

This property will not be able to block a very well distributed denial of service attack. If each ip address only has one pending connection, then this property will not be able to block the attack, but it will still detect it because the total number of pending connections will still be greater than the total number of open connections.

The compilation of this property is easier than the rest of the properties because it was written directly in LARVA and thus no translations were required.

9.3.3 Evaluation

There are many other kinds of attacks which can be monitored. Other properties would have to be written in order to protect against these threats. As usual, the trade-off is high level of security against the overhead incurred by monitoring. However, one would normally prefer to incur a little slowdown in the system and be able to automatically monitor and possibly mitigate any attacks on the system. A little slowdown is acceptable even more because of the guarantee that the properties being verified in this case study are all slowdown truth preserving.

One should also keep in mind that the current LARVA implementation is more of a proof of concept rather than an industrial system. The implementation of clocks and channels uses threads and the overall monitoring infrastructure uses a considerable amount of memory. Having said this, we believe that there is a lot of room for improvement on the implementation level and more attention can be given to efficiency and more careful use of resources.

One simple approach to go about the problem of resource consumption is to write the properties in Lustre. This is much more efficient because communication is done using variables rather than channels and it only uses a single global clock rather than an arbitrary number of clocks. Furthermore, the upperbound of memory and temporal requirements for a Lustre program can be calculated at compile time. The limitation is that timer events are not available in Lustre.

Another possible approach to provide extra processing resources is to perform the monitoring on a separate machine and send the events generated to this machine. Even with this configuration there may still be insufficient resources to be able to monitor all the events instantly. The practical solution is to adapt according to the situation by making best use of the available resources without adversely affecting the system's usability.

9.3.4 Conclusion

Using LARVA architecture as a means of network monitoring has proved to be relatively easy and effective. The advantage is that different notations can be used according to the need. Furthermore, these different notations offer different expressive powers and also different guarantees regarding the effect of the monitoring on the system. Another positive aspect is that the network monitoring system can be configured not only to detect attacks but also to take action against such attacks as in the last property.

9.4 Conclusion

Both case studies have been positive in their respective ways: the first from an industrial and very practical aspect, the second as a proof of the usefulness of real-time properties in desktop applications. The first case study gave us an idea of what to expect when trying to apply a runtime verification on a third party system with no planning for runtime verification. This gave us insight into how runtime verification can be more easily integrated in the process of software development. The first case study also helped us understand the required expressivity of Larva. For the second case study, the Larva architecture was fully developed with all the necessary translation algorithms. The system monitored was created for the purpose of monitoring and therefore we experimented with different notations and showed the usefulness of runtime monitoring of real-time properties.

10. Comparison to Other Work in Runtime Verification

10.1 Introduction

In this chapter we will consider logics and automata which have been used for runtime verification. These will be classified under a number of headings in Section 10.2. For logics and automata to be useful, they are incorporated into tools. Larva is one such tool which we have created. In Section 10.3 the tools used for runtime verification are compared in a benchmark. The chapter is concluded in Section 10.4.

10.2 Logics and Automata

The literature is full of proposed formal notations for specifying system properties. Different authors have suggested different specification notations according to the particular domain, trying to improve one aspect or another of the existing notations. An important trade-off that should be highlighted is that the greater the expressivity of the formal notation used, the more complex is the algorithm for its verification.

10.2.1 Meta-languages

Some of the logics which have been proposed are actually meta-languages which allow the user to specify other logics. For example, in the case of Eagle [10] and Maude [22], the proposed architecture provides the basic temporal logic constructs and then allows the user to create his own domain-specific logic.

We investigated two meta-languages: Maude and Eagle. Maude [23, 21] is a metalanguage interpreter which supports equational and rewriting logic computation based on the principle of reflection. Thus, during execution, the specified logic of a system is concurrently rewritten until, possibly, a fully evaluated value is reached. From a computational point of view rewriting rules can be considered as transition rules. From a logical point of view they can be considered as inference rules. The rewriting process is in fact continually modifying the system of properties along the execution path — hence the term reflection. This has successfully been used in the runtime verification tool Java PathExplorer [52].

Eagle [10, 44] is a runtime verification tool comprising a rule-based language and an interpreter for it. This has been specifically designed to support future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints, and statistics. To define this variety of logics and constraints, Eagle only requires the basic logical operators and two primitive temporal logic operators *next* ($\bigcirc$) and *previous* ($\odot$). Other operators can then be expressed using these primitive operators. Eagle is implemented as a Java library and also allows parametrisation of rules. This means that if one would like an acknowledgement to be associated with a particular sent message, the message id can be used to distinguish the corresponding acknowledgement. This provides the logic with expressive power which propositional temporal logic does not have. Another advantage of Eagle is that a lot of its properties can be expressed in state machines which users tend to find more intuitive. Each transition has a condition and an action on the variables of the state. The condition is not only based on the input of the state machine but also on the variables which constitute the state of the machine. Furthermore, the rules expressed in Eagle, can be either maximal or minimal fix-point semantics. This allows more flexibility in expressing weak and strong versions of the same operators. To make this clearer, we will illustrate it using an example. We will define a weak and a strong version of the *until* operator ($\mathit{Until}(F1, F2)$ means that $F1$ has to hold until $F2$ holds). The following is the strong version (which is not satisfied unless $F2$ becomes true):

$$\underline{\mathit{min}}\ \mathit{Until}(\mathit{Form} F1, \mathit{Form} F2) = F2 \vee (F1 \wedge \bigcirc \mathit{Until}(F1, F2))$$

The weak version is the same but with $\underline{\mathit{max}}$ instead of $\underline{\mathit{min}}$. All these features make Eagle very versatile for runtime verification.

Hawk [27] is a programming-oriented extension of the Eagle logic. It provides the means to capture program events with parameters which can be used in Eagle formulas. Parameters can be executing threads, the objects that methods are called upon, arguments to methods, and return values. Hawk is also able to create a monitor for a given formula and automatically instrument it to the target system.

10.2.2 Tool-Specific Logics and Specification Languages

Some systems also propose their own specific language. These have been investigated so that we get an idea of what other designers have found useful to include in their languages. The following is the list which we have explored:

- A simple language designed by Fradet and Hong Tuan Ha [38] is easily transformed into timed automata. A particular feature of this language is that it allows the specification of real-time properties. It also uses aspect-oriented programming to weave the verification with the actual system being monitored. An example of how the architecture works is as follows:

Consider the following two lines:

```
a1 = M.get \rhd start(t); a2
a2 = M.get \rhd \{wait(t, 20); start(t)\}; a2
```

M.get simply means that if the method *M.get* is called, then, the system enters the first state $a1$. Consequent, a timer t is started and the system moves to state $a2$. This time, if *M.get* is called, we have to *wait* for 20 seconds, and then the timer t is re-started and the state machine goes once more to $a2$. The architecture also supports other timer functions such as *reset* and *cancel*. Another interesting feature is that from one state, we can have a possibility of going into more than one state by adding other entries starting with a specific method call and followed by a clock operation. However, it is important to note that it is assumed that the possibilities are exclusive so that the system remains deterministic. Another suggested way of keeping the system deterministic (but not yet implemented) is by giving priority according to the order in which the transitions are listed.

- The Policy Specification Language (PSLang) [34] is used in the Policy Enforcement Toolkit (PoET). In PSLang, the security state is stored in named and typed variables. This makes the system more transparent to the user because there is no hidden variables of which the user is not aware. Furthermore, it is able to use low-level actions to synthesize higher-level security events according to the specified policies. Consequently, these events can trigger the required enforcement activities. This hierarchy of actions and events makes it possible to make policies reusable and more clear without unnecessary details. PSLang has been specifically designed to be easily used and thus the syntax and semantics are based on Java and JVML.

- ConSpec [2] is inspired by PSLang, but restricted to mobile devices with limited resources. It is assumed that a contract is defined for each application. Upon installation of an application, the contract is checked against the policies of the device on which it is being installed. If the application's contract does not comply with the device's policies, the application cannot be installed on the device. In other cases, where the application's contract cannot be definitively checked before installation, a runtime monitor is inlined to the application.

- Java-MOP [20] is a monitoring-oriented development environment for Java. The motivation behind this paradigm is to combine the specification and the implementation of a system. It is different from runtime verification because the monitoring system is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. For example, Chen et al. [18] give an example of a system which does user authentication using a security policy in monitoring-oriented programming. An appealing feature of Java-MOP is that it can be extended with different logics including FTLTL, PTLTL, ERE and Jass.

- Polymer [12] was developed to help users enforce security policies on untrusted Java applications. A Polymer policy is implemented by extending the *Policy* object which contains decisions (queries) and actions, security state of the running application and methods to update the policy's security state. Furthermore, policies can be used to compose higher-order security policies which in turn will combine the sub-policies in a semantically meaningful way.

- Tempura [73] has been proposed specifically to describe interval temporal logic. An example of a Tempura formula is as follows: $(M = 4) \wedge (N = 1) \wedge \mathit{halt}(M = 0) \wedge (MgetsM - 1) \wedge (Ngets2N)$. This means that in the current state, M should be equal to 4, and N should be equal to 1. Then for the interval to satisfy the formula, in the next state, M should be equal to 3 (because $MgetsM - 1$) and similarly N should become 2. In the subsequent states the constraints on M and N should hold until the *halt* condition is met. If all the states within the interval satisfy the resulting values of M and N, then the interval satisfies the formula.

- The Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) used in the Monitoring and Checking (MaC) architecture [67] are two complementary languages which allow for a clear separation between the definition of primitive events (using PEDL) and system properties (using MEDL).The motivation for this separation is that it makes the system easier to adapt to other programming languages, leaving the higher level definition language (MEDL) intact. For example, using PEDL we may specify a primitive event as follows:

  ```
  Event OpenGate = StartM( GateController.open() );
  ```

  Then, using MEDL we write:

  ```
  Cond GateClosing = ( CloseGate when !Gate\_Down, OpenGate )
   => lastClose + 30 > currentTime;
  ```

  The example is a simplified version of the one given by Lee et al. [67]. It checks that the time being taken by the gate to close is not longer than 30 time units.

An implementation of the Monitoring and Checking architecture for Java is Java-MaC [63]. Using Java-MaC involves the following steps: (i) the PEDL and MEDL scripts are compiled for the event recogniser and the runtime checker respectively; (ii) the program is automatically instrumented to gain access to the system events; (iii) instrumented programs send an event stream to the event recogniser; (iv) the event recogniser identifies the higher-level activities; (v) the event recogniser communicates the recognised events to be processed by the runtime checker — raising an alarm if any of the specified properties are violated.

- Lola [28] is a synchronous language which allows the user to specify the properties of a program in past and future LTL. The advantage of Lola is that as a synchronous language it guarantees bounded memory to perform online monitoring, but differs from most other synchronous languages in that it is able to refer to future values in a stream. This is not possible in other synchronous languages which can be executed — Lola is descriptive rather than executable. Lola is similar to Eagle in its expressivity, and both past and future LTL are supported in Eagle, but they differ in their descriptive nature. Lola allows the user to collect statistics at runtime and to express numerical queries. Triggers can also be defined in Lola to generate notifications when a particular boolean expression becomes true.
- PSL (Property Specification Language) [32] allows the user to specify system properties which are mathematically rigorous and automatically verifiable. An interesting aspect of PSL is that it provides two versions of the temporal operators — the weak and the strong. Sometimes during monitoring it is not possible to decide whether a property holds or not because only a part of the trace is available at runtime. For example, consider the rule $eventually(p)$ in a trace being executed where p has not yet occurred. It is considered to hold in the weak version and violated in the strong version. Furthermore, PSL also includes other features such as Sequential Extended Regular Expressions and clocks.

10.2.3 Other Logics

- We will first start by the basic linear temporal logic (LTL) proposed by Pnueli (as reported in [65]). In LTL we define properties which hold on a sequence of states without branching (unlike computational tree logic. This is ideal when we consider a single execution path as in runtime verification. An example of a property which can be specified in LTL is $G(p)$ ($globallyp$) which states that p should hold for all the states in the trace. We can restrict LTL to consider only past states in an execution trace. The resulting logic is known as past time LTL. Similarly, we can consider only the future states using future time LTL. In various cases it is much more convenient to express certain temporal properties using past time LTL rather than future time LTL [70], even though they have the same expressive power [40]. Another variation of LTL is finite trace LTL [54]. This is an adaptation of LTL which is applicable to a trace with a finite number of states rather than an infinite trace. This is especially useful for runtime verification because only a part of the execution trace is available at runtime.

The metric temporal logic (MTL) [6]was introduced as an extended LTL which can represent real-time properties. MTL with some extensions has been successfully used in Temporal Rover [30]. Basically, it uses the same notation as LTL and adds real-time quantities as additional constraints. For example, the MTL property $\Box_{[0,5]}p$ means that p must hold throughout the interval $[0,5]$. Note that this property is the same as the LTL property $G(p)$ but applied to particular interval.

- Regular expressions have been suggested as an extension to the MEDL language [83] in the MaC architecture. The motivation of adopting regular expressions is that they are more convenient to express certain complex orderings of events. In fact, a considerable number of temporal logics and architectures have been specifically designed or extended to include regular expressions' expressiveness. Examples include: ForSpec Temporal Logic (FTL) [7], the logic supported in monitoring-oriented programming [19], PSL [32]. Other logics such as QDDC [75] have been shown to be able to encode regular expressions. A very positive aspect of regular expressions is that they are used for other applications such as string matching. Thus, developers are familiar with the way patterns can be represented by regular expressions. For example imagine we want to denote a simple rule: any number of occurrences of event a can occur before event b. Using regular expressions this can be written simply as: $(a^*)b$. Using another logic such as linear temporal logic it can be written as:
$$\mathit{AsThenB}(\varphi_n) \stackrel{\text{def}}{=} b \lor (a \land \mathit{Next}(\mathit{AsThenB}(\varphi_{n-1})))$$
The latter can be much more difficult to come up with, especially for someone who is not accustomed to temporal logics. However, regular expressions also have their drawbacks and may not be intuitive even for someone who is accustomed to use them in other domains. For example, upon seeing a regular expression made up of system events, the user may get confused whether the represented sequence is an accepted or a non-accepted sequence of events.

- Other logics have already been introduced in detail in Chapter 3. These include duration calculus, QDDC, and counterexample traces.

Automata in Verification

Automata have been extensively used in verification especially for efficiently deciding whether a property is violated at a particular state. A very attractive advantage of using automata is that they have a pictorial representation. This representation may be more intuitive than other textual representations of security properties. Another advantage of automata is that a lot of notation conversions (see the following list) use automata and therefore, automata may be an ideal way of integrating different notations together.

- In the case of regular expressions as used with MEDL [83], they are first converted into a finite state automaton. This is then used for the actual verification algorithm. Finite state automata are proposed [81] to efficiently monitor temporal properties written in Allen temporal logic. Automata are also proposed to check finite traces [43].
- Büchi Automata (as used by Courcoubetis et al. [26]) has been suggested to check finite traces by generating the corresponding labelled generalised Büchi automaton (LGBA) from LTL [29, 42]. A similar approach is used in [14] and [26] where the Büchi automata are used for efficient verification of temporal properties. Also, the properties specified in the model checker SPIN [59] are converted to Büchi automata. Subsequently, the whole system together with the specified properties becomes the asynchronous interleaving product of automata. However, a number of problems have been identified [82] which arise from the use of Büchi automata. Most notorious are the problem of converting LTL formulas into Büchi automata and secondly, the problem of checking finite traces when Büchi automata are thought to handle infinite strings. This explains why a number of other verification systems [82, 67, 30] do not use Büchi automata.
- Alternating Finite Automata (AFAs) (as used by Finkbeiner and Sipma [37]) have also been very commonly suggested for the verification of temporal properties [31, 86, 37]. The advantage of using AFAs is that it has *and*-states and *or*-states which can be exploited to represent the recursive definitions of LTL which also involves *and*s and *or*s. Another advantage is the visual appeal of AFAs [31] and more importantly, that AFAs are linear in size to the corresponding LTL formula [37]. Once we have the AFA equivalent of the LTL formula, the AFA is reconfigured according to each online input, minimising the AFA where appropriate.
- Timed automata [3, 4] have been repeatedly proposed for verifying real-time temporal properties [38, 85, 15]. On the transitions, apart from an event, we can define clock operations. Figure 10.1 shows a simple example of a timed automaton.

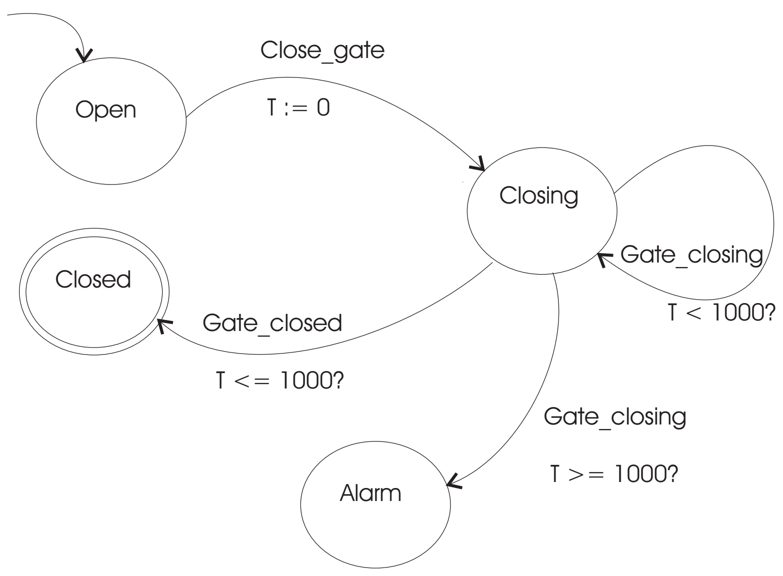


Figure 10.1: An example of a timed automaton which handles the closing system of a gate.

It represents the logic required to ensure that a gate closes within 1000 time units. Thus, as soon as a *Close_gate* event is received, a timer is reset to zero. If 1000 time units elapse and the gate is still open, then we proceed to an alarm state. If the gate is closed within the given time, then we proceed to an accepting state.

The literature contains various applications where timed automata where used. In [38], the architecture involves the translation of both the system and its properties into separate timed automata. Subsequently, these are both weaved together to produce a single timed automaton. There are other examples of applications of timed automata [85, 15]. Interestingly, Bouyer [15] adds weights as an extension of timed automata with costs. This is especially useful for simulating resource consumption in timed systems.

- Mode automata [69] have the purpose of providing a two-level automaton (multi-level if a composition is used). The motivation behind this structure is the need to separate logic which runs at a particular instant (say only during take-off) from other logic that runs at another instant (say only during landing). The need of separation occurs frequently in many applications. For example, mode automata may prove to be very useful in separating the verification of states in a transaction from the internal logic necessary for each state. Hence each state represents a possible mode in which we can be.

10.3 A Benchmark

A Java program representing a bank processing a number of transactions for a number of users has been developed to experiment with the use of the different systems. The reason for using such a system is two-fold. Firstly, a number of interesting properties can be specified easily. Secondly, a bank system can be easily understood because we are familiar with bank systems. In our bank system there are a number of users and each user can have a number of transactions. Moreover, the bank system has a database which is used to simulate communication and time delays in the system. When a transaction is executed there are three possible results: success, failure and exception. Upon a failure, the transaction is successively retried for another four times. No retries are performed in case of an exception. Note that the intention of the benchmark case study is primarily to compare property specification and monitoring.

We have identified a number of classes of properties, and concrete examples for the bank processing system, to compare and contrast the use of the different tools.

Scope: The type of scope which can be specified. Types of scope include object (denoted by O in Table 10.1), session (denoted by S in Table 10.1) — one run of the application, multisession — current and previous runs, global — all running applications of a system. This is used to specify on which level the property is verified. For example if the scope is 'object' then the property will be verified for each individual object.

Exceptions: Exception handling and throwing in an application usually represent important events in a system. This aspect represents whether or not the user can express properties which include exception throwing and handling.

Real-time: Real-time refers to whether or not the monitored properties can include real-time. This means that the verification system is able to trigger checks at particular time intervals and compare clock values upon particular system events.

Invariants: We use the term invariants to refer to inbuilt mechanisms in the verification system to monitor the changing of values of variables. The purpose is to be able to verify that certain variables only change when they are supposed to do so.

Feedback: It refers to the capability of the monitoring system to return feedback to the target system. This usually takes the form of a mitigation action in case a violation is found. In other cases this may be limited to stopping the program's execution (denoted by St in Table 10.1).

Table 10.1: Expressivity features of various tools.

Tool	LARVA	ConSpec	MOP[a]	MaC[b]	Hawk	Lola
Scope	*S/O*	✓[c]	*S/O*	*S*	*S*	*S*
Exceptions	✓	✓	×	×	×	×
Temporal Logics	×	×	✓	×	✓	✓
Real-Time	✓	×	×	✓[d]	✓[e]	×
Mob. App. Policies	×	✓	×	×	×	×
Invariants	✓	×	✓	✓	×	×
Feedback	✓	*St*	✓	✓	×	×
Conditions	✓	✓	✓[f]	✓	×	×
Numerical Queries	×	×	×	×	×	✓

[a]Java-MOP
[b]Java-MaC
[c]in specification it supports all the mentioned scopes but currently only *session* is supported
[d]restricted (cannot trigger clock events)
[e]can be extended to support real-time
[f]restricted to implementing conditions in violation/validation handling method

Conditions: This refers to the ability to filter events by applying a condition on the parameters and/or monitoring variables.

Temporal logics: It represents the fact that the tool supports specification written in temporal logics such as LTL.

Mobile application policies: We refer to the ability of defining a security policy which can be partially verified before runtime if the application also specifies its policy. Verifying applications for mobile devices require the monitoring system to be as lightweight as possible.

Numerical queries: This refers to explicit support to expressing numerical queries about statistics of the program being verified.

Table 10.1 shows which tool have *explicit* support for the aspect being considered. Note that the meaning of the scope object is sometimes referred to as class. In the case of Larva, the same object need not necessarily be the same instance, but be equated through the (optional) use of a user-provided equality method. One advantage of this approach is that when monitoring objects which are serialised and de-serialised, the object before serialisation will still be considered the same as the object afterwards (even though they are not the same instance).

Although with its own limitations, Larvacan express a number of interesting classes of properties, not all of them expressible directly in the other tools. Two limitations of Larvaare that it cannot support different temporal logics (Hawk, Java-MOP and Lola do have this capability), and it is not suitable for security of mobile devices (where ConSpec excels).

Table 10.2: Larva overheads for the benchmark example.

Test Reference Number	0	1	2	3	4
System without Monitoring					
time(ms)	4	4	6303	3	9
memory(Kb)	23	23	23	70	161
System with Monitoring					
time(ms)	123	120	6395	161	176
memory(Kb)	453	209	160	467	434
System with Monitoring without Clocks					
time(ms)	55	60	n/a	n/a	36
memory(Kb)	432	477	n/a	n/a	378

10.3.1 Performance of Larva

Five tests have been built for the evaluation of the performance of Larvain terms of overheads. *Test 0* executes a number of transactions but does not violate any of the given properties. Subsequently, *Test 1* violates the invariant property by trying to change the transaction amount. *Test 2* violates the property that a retry should occur within two seconds. *Test 3* violates the property the a user cannot have more than five transactions. Finally, *Test 4* violates the property that upon an exception the transaction is not retried.

Table 10.2 shows statistics for the benchmark when the five tests were run under three different configurations: (i) without monitors; (ii) with monitors for all the properties; (iii) removing the monitors which include clocks with the purpose of investigating the impact of clocks on the monitoring system.

The numbers presented may be a bit distorted because of the operating system scheduler and the Java garbage collector. If the garbage collector is called before measuring the used memory, the numbers would be much smaller than the ones presented in the following tables.

Although the resources required for the program to run without monitors as compared to when it was run with monitors seems huge, the overhead is linear in the size of the automaton used to describe the properties. For example compare 4 milliseconds to 123 milliseconds and 23 kilobytes to 453 kilobytes for *Test 0*. Also, one should notice the impact of having clocks in the program execution time. For example compare 55 milliseconds to 123 milliseconds. However, if we analyse the figures in more depth we realize that the situation is not so dull and grim. The reason being that the monitors were added to a dummy system with empty methods and empty objects. Therefore it is not surprising that the overhead of monitoring seems to be so relatively large. What we did to proof our intuition that the overhead seems unrealistically large is to increase the size of the system with regards to the required memory and processing time preserves the complexity as shown in Table 10.3, in which *Test 0* is used.

Table 10.3: The statistics obtained by trying *Test 0* on variations of the benchmark.

Test Variation	Normal	Time Cons.	Big Obj.	Many Obj.
System without Monitoring				
time(ms)	4	4722	4874	53849
memory(Kb)	23	23	260	2384
System with Monitoring				
time(ms)	123	5321	5458	65153
memory(Kb)	453	418	458	3509

Table 10.4: The benchmark statistics for various tools.

Test Ref. No.	0	1	4	0	1	4
LARVA[1]				ConSpec		
time(ms)	27	23	30	7	n/a	n/a
memory(Kb)	91	136	208	54	n/a	n/a
Java-MOP				Java-MaC		
time(ms)	23	23	52	7	n/a	n/a
memory(Kb)	174	173	312	26	n/a	n/a

The first experiment was to increase the processing time which the system without monitors require to complete the execution. One should notice how the gap between 4722 and 5321 milliseconds is relatively much smaller than the gap between 4 and 123 milliseconds. The second experiment was to increase the amount of memory which each object requires. In this case the total memory used was 458 kilobytes which is very close to the memory initially used for the initial experiment (453 kilobytes).

In order to investigate the real relationship between the size of the monitoring system and the monitored system, we multiplied the number of monitored objects by 10. The result obtained substantiates the intuition that the size of the monitor is directly proportional to the number of monitored objects.

It is difficult to have a true comparison among the tools since they do not have the same expressive power. Therefore, we could not implement all the monitors for all the tools. For example none of the other tools handle real-time properties. For this reason we intentionally removed the monitors for these properties from our program. Another interesting issue is that LARVA issues a report regarding the status of the monitoring system. This is not done by other tools. Therefore, we modified our code and removed the logging system so that the comparison is done on level ground. The results obtained are shown in Table 10.4. Note that Hawk and Lola are not in Table 10.4 (nor in the subsequent tables) since we did not have access to the tools. The results concerning their expressiveness were based on descriptions of the tools from research papers and personal communications.

The most similar tool to Larva is undoubtedly Java-MOP since it can implement the same properties in a very similar fashion and both Larva and Java-MOP use AspectJ as the underlying framework. Compared together, the statistics show that there is little difference. ConSpec is restricted to security properties on mobile devices so we could not go further in our comparison. To give an idea of how much resources ConSpec uses, we implemented the property which limits the number of users rather than the number of transactions per user. This explains why the time and memory required were much less than Larva and Java-MOP. Finally, we tried Java-MaC. Again, it is difficult to compare the results because none of the properties could be implemented in Java-MaC. Furthermore, Java-MaC uses a different technology – it transmits the event stream to other applications running simultaneously. These factors explain the difference in the amount of resources used.

10.4 Conclusion

In this chapter, we have first given an overview of the work done in the field of runtime verification. Most notably, we have considered a number of logics which have been proposed over the years for various runtime verification architectures. Subsequently, we have compared Larva to other similar architectures. This comparison has the purpose of highlighting the various levels of expressivity according to a number of criteria. The result of this study shows Larva to be a highly expressive logic. Towards the end of the chapter, we have also used a benchmark to compare the resource utilisation of different architectures. In this regard, when compared to tools with similar expressivity, the performance of Larvais good. However, one should note that the current implementation of the tool has not been optimised.

Part V

Conclusions

11. Conclusion

In this chapter, we will start off by giving a summary of the work presented in this thesis, followed by an exploration of the possible future direction of this research. Furthermore, we will propose numerous possible extensions to the work presented in this thesis. The final section will conclude the work with some final remarks.

11.1 Summary

With the ever growing need of robust and reliable software, formal methods are increasingly being employed. A recurrent problem is the intractability of exhaustively verifying software. Due to its scalability to real-life systems, testing has been used extensively to verify software. However, testing usually lacks coverage. Runtime verification is a compromise whereby the current execution trace is verified during runtime. Thus, it scales well without loss of coverage. Substantial work has been already done in the area of runtime verification. Thus, we first presented an overview of the work carried out so far so that our work can be placed into perspective. A particular area of interest is the specification of real-time properties. There are various logics whose underlying model of time varies from points in time to continuous real-time. After presenting some of the logics available, we have argued in favour of a clear and succinct logic which is able to express a wide range of properties. Such a logic would include the expressivity to represent properties with context, invariants, and real-time amongst other things. This was the motivation behind the creation of the DATE logic. DATEs are dynamic automata with timers and channels. Being dynamic, DATE automata can be replicated for each active object to monitor properties with context. Using timers, they can monitor real-time properties. Channels allow DATEs to remain modular such that each automaton can perform a single specialised task. DATEs are triggered by system events and are also able to send back feedback to the system according to the property being monitored. For this logic, we created the Larva architecture. This includes a language specification for the description of DATEs and a compiler which translates the Larva script into the necessary Java code. The Larva script is designed to be as clear and succinct as possible, while at the same time maintaining the necessary expressivity. The compilation of the script generates AspectJ code — to capture useful system events — and Java classes which implement the automata described in the script.

We must admit that DATE is not the most appropriate logic to represent all the possible properties of software. Sometimes, it is easier to represent a property in some other logic such as duration calculus. For this reason, we created a translation from a subset of duration calculus, called counterexample traces, into LARVA. Thus, our architecture can still be used while at the same time expressing properties in another, possibly more appropriate logic. Similarly, a translation was devised the language, Lustre. Its advantage is that both the memory and temporal requirements of a Lustre program can be calculated at compile time. Another useful subset of duration calculus is QDDC. This can be translated into Lustre. Therefore, using QDDC, one will have both the advantage of a more appropriate logic for particular cases and the guarantees on the size of the overhead given by Lustre. Finally, we also have a translation from the contract language $\mathcal{CL}$ into LARVA. This allows contracts to be monitored by LARVA.

Monitoring introduces an overhead on the system and thus slows it down. This is a serious issue in monitoring real-time properties. It is virtually impossible to remove this overhead, but guarantees can sometimes be given regarding the impact of the overheads on real-time properties. Some real-time properties which are satisfied will not be violated by a slowdown of the system. We call these properties slowdown truth preserving properties. Similarly, there are speedup truth preserving properties which are not violated when the system is speeded up. By considering fragments of duration calculus, we have shown which fragments preserve truth on slowing down or speeding up a system. If one uses slowdown truth preserving properties, we can guarantee that the properties which are satisfied cannot be violated by introducing monitors. Similarly, for speedup truth preserving properties, we can guarantee that the properties which are satisfied cannot be violated by removing monitors.

As a test of the usefulness of LARVA, we tried it on two real-life case studies: an industrial system which handles financial transactions and a network intrusion detection system. LARVA was found to be useful in both cases and interesting properties were formulated and verified. Finally, we have compared LARVA to other similar tools on two aspects: expressivity and resource consumption. As regards expressivity, a number of aspects were selected and we showed whether each tool is expressive in each aspect. As regards resources, we set up a benchmark with a number of typical but non-trivial properties and a number of tools were used to represent these properties. It was found that LARVA compares well with similar tools.

11.2 Future Work

11.2.1 Off-line Runtime Verification

If the overhead of runtime verification is too large for the available resources, the best alternative may be that verification takes place off-line — the properties are not checked synchronously as they occur, but some time later when there are resources available. This has the disadvantage that any mitigation actions cannot take place instantly. There are, however, numerous applications where a few seconds or minutes delay is perfectly acceptable. For example, consider a rollback in a database; if the rollback occurs a few seconds after an error is identified, this still keeps the integrity of the database. In such cases, the verification of properties can be carried out on a separate machine, with the possibility that the machines do not keep up with each other. For this reason there needs to be some kind of protocol by which the machines can, for example, decide to synchronise, or raise an alarm if the discrepancy is more than a particular amount of time. A possible feature may be that, if the target system enters a critical section, the monitoring system will synchronise, leaving less priority verification for later.

11.2.2 Extensions to Logics/Languages

A possible extension to counterexample traces and phase event automata would be to further consider the introduction of counters. Their behaviour will be similar to that of clocks but instead of measuring real-time, they measure event occurrences. In a number of practical situations, counting events may be very useful to detect malicious usage patterns. A case in point is the intrusion detection system in the second case study.

Another improvement may be to extend the Lustre language to include timers. In this work we simply pass a timestamp as a parameter which allows us to verify a number of interesting real-time properties. However, this has the limitation that the system cannot perform any steps without an external event. A timer should be able to trigger the verification so that as time elapses the system can automatically detect a bad state, without waiting for an external event.

11.2.3 Extensions of the Theory

The theory developed in this work can be extended to other logics. For example the theory of slowdown/speedup truth preservation, which was so far developed for duration calculus, can be extended to QDDC. Similarly, the work on memory upperbounds, which was so far developed for QDDC, can be possibly extended for phase event automata.

Thus far, in our work about slowdown/speedup truth preservation, we have always assumed that all state variables are affected by the same time function. However, this is not always the case. For example, in a distributed real-time system the state variables on different machines/threads may have a different time function. In other words, the slowing down on one thread may be different from another. This is much more challenging than simply considering the affect of one time function on a single machine/thread.

11.2.4 Industrial Applications

On a more practical level, it would be very useful if more real-life case studies are carried out. With more experience in the field, the research community would be in a better position to understand the needs of the industry and it would be easier to gain more confidence from the industry. More experience would also contribute towards the maturity of the field and software engineers will be more willing to integrate runtime verification with the software development life cycle.

11.3 Concluding Thoughts

The vast literature in the area of runtime verification is a proof of the growing interest in the subject. Nevertheless, the subject is still relatively new and there is a lot of space for research. The ideas which we have proposed so far have already brought up considerable interest from the local industry and this is a very positive sign. We are very optimistic that the research will evolve with a lot of more innovative ideas and possibly provide practical solutions for current security issues.

A. Publications

The work presented in this thesis includes the work presented in the following published and unpublished papers. The contribution of the respective authors is explained below.

Published Papers

1. Christian Colombo, and Gordon J. Pace. Aspect-Oriented Programming Runtime-Enforcement of Temporal. In CSAW 2007: Proceedings of the Computer Science Annual Workshop, pages 150-161, Malta, 2007.

   Pace reviewed the document and rewrote parts of it.

2. Christian Colombo, Gordon J. Pace and Gerardo Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties, FMICS 2008, Italy.

   Pace wrote Section 2.1 and reviewed the document and rewrote parts of it, Schneider reviewed the document several times rewriting parts of it.

3. Christian Colombo, Gordon J. Pace and Gerardo Schneider. A Practical Approach to Runtime Verification of Real-Time Properties for Java Programs, WICT 2008, Malta.

   Pace reviewed the document and rewrote parts of it, Schneider reviewed the document.

Papers in Progress

1. Christian Colombo, Gordon J. Pace and Gerardo Schneider. Monitoring Slowdown and Speedup Truth Preserving Real-Time Properties at Runtime, 2008.

   Schneider wrote the introduction and reviewed the document rewriting parts of it. Pace wrote Section 2.4 and the majority of Section 3. He also reviewed the document and rewrote parts of it.

References

[1] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 196–210, Liege, Belgium, 1995. Springer Verlag.

[2] I. Aktug and K. Naliuka. Conspec: A formal language for policy specification. In *FLACOS '07*, pages 107–109, Oslo, Norway, October 2007.

[3] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur and T. A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[6] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993.

[7] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[8] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.

[9] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[10] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.

[11] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Berlin, Heidelberg, Nov. 2007. Springer-Verlag.

[12] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.

[13] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Trans. Software Eng.*, 28(2):129–145, 2002.

[14] E. Bodden. Efficient and expressive runtime verification for java. Grand Finals of the ACM Student Research Competition, 2004/2005.

[15] P. Bouyer. Weighted timed automata: Model-checking and games. *Electronic Notes in Theoretical Computer Science*, 158:3–17, 2006.

[16] G. Chakravorty and P. Pandya. Digitizing interval duration logic. In *Proc. CAV 2003*, Colorado, Boulder, July 2003. (Technical Report, TCS-02-PKP-1, Tata Institute of Fundamental Research, 2002).

[17] Z. Chaochen and M. R. Hansen. Chopping a point. In *BCS-FACS 7th Refinement Workshop*. Electronic Workshops in Computing, Springer-Verlag, 1996. http://www.springer.co.uk/ewic/workshops/7RW/.

[18] F. Chen, M. D'Amorim, and G. Roşu. Monitoring-oriented programming: A tool-supported methodology for higher quality object-oriented software. Technical Report UIUCDCS-R-2004-2420, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 2004.

[19] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV'03*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[20] F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The maude system. In *RTA*, pages 240–243, 1999.

[23] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. *Electr. Notes Theor. Comput. Sci.*, 4, 1996.

[24] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.

[25] S. Colin and L. Mariani. *Model-Based Testing of Reactive Systems*, volume 3472. Springer Berlin / Heidelberg, 2005.

[26] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[27] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[28] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME'05*, pages 166–174. IEEE Computer Society Press, June 2005.

[29] R. Deutschmann, M. Fruth, H. Reichel, and H.-C. Reuss. Trace checking with real-time specifications. In *Proceedings of the 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)*, Braunschweig, Germany, December 2004.

[30] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[31] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. *Journal of Universal Computer Science*, 12(5):482–498, 2006.

[32] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[33] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *cav03*, volume 2725 of *lncs*, pages 27–39, Boulder, CO, USA, July 2003. springer.

[34] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, 2004.

[35] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9 2003.

[36] S. Fenech. Conflict analysis of deontic conflicts. Master's thesis, University of Malta, 2008.

[37] B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, 2004.

[38] P. Fradet and S. H. T. Ha. Systèmes de gestion de ressource et aspects de disponibilité. *Revue francophone L'Objet*, 12(2-3), 2006.

[39] D. Gabbay. The declarative past and imperative future. pages 35–67, 1996.

[40] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM.

[41] A. Gal, O. Spinczyk, and W. Schröder-Preikschat. On aspect-orientation in distributed real-time dependable systems. In *WORDS*, pages 261–270, 2002.

[42] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[43] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering*, 2001.

[44] A. Goldberg and K. Havelund. Automated runtime verification with eagle. In *MSVVEIS*, 2005.

[45] L. Gonnord, N. Halbwachs, and P. Raymond. From discrete duration calculus to symbolic automata. *Electr. Notes Theor. Comput. Sci.*, 153(4):3–18, 2006.

[46] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[47] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

[48] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and

G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[49] M. R. Hansen and Z. Chaochen. Duration calculus: Logical foundations. *Formal Asp. Comput.*, 9(3):283–330, 1997.

[50] M. Hatton. *Mathematical Analysis.* Hodder and Stoughton, 1977.

[51] K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag.

[52] K. Havelund and G. Roşu. Java pathexplorer — a runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 18–22 2001.

[53] K. Havelund and G. Roşu. Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[54] K. Havelund and G. Roşu. Testing linear temporal logic formulae on finite execution traces. Technical report, RIACS, 2001.

[55] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[56] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.

[57] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 545–558, London, UK, 1992. Springer-Verlag.

[58] J. Hoenicke. *Combination of Processes, Data, and Time.* PhD thesis, University of Oldenburg, July 2006.

[59] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[60] S. Jalili and M. MirzaAghaei. Rverl: Run-time verification of real-time and reactive programs using event-based real-time logic approach. In *SERA '07:*

Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007), pages 550–557, Washington, DC, USA, 2007. IEEE Computer Society.

[61] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[62] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241*, 1997.

[63] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[64] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[65] L. Lamport. "sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

[66] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[67] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[68] M. Leucker and C. Scallhart. Monitor-based runtime reflection. In *FLACOS'07*, 2007.

[69] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[70] N. Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, 79:122–128, 2003.

[71] B. Maszkowski and Z. Manna. Reasoning in interval temporal logic. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 371–382, London, UK, 1984. Springer-Verlag.

[72] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[73] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, 1986.

[74] O. Ochoa, I. Gallegos, S. Roach, and A. Gates. Towards a tool for generating aspects from medl and pedl specifications for runtime verification. In *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, 2007.

[75] P. Pandya. Specifying and deciding qauntified discrete-time duration calculus formulae using dcvalid. In *Proc. Real-Time Tools, RTTOOLS'2001 (affiliated with CONCUR 2001)*, Aalborg, August 2001. (Technical Report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000).

[76] P. K. Pandya. Interval duration logic: Expressiveness and decidability. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[77] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2007.

[78] A. P. Ravn. *Design of Embedded Real-Time Computing Systems.* PhD thesis, Technical University of Denmark, October 1995.

[79] P. Raymond. Recognizing regular expressions by means of dataflow networks. In *ICALP '96: Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, pages 336–347, London, UK, 1996. Springer-Verlag.

[80] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.

[81] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic –translation to ltl and monitor synthesis–. In *Proceedings of 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[82] G. Roşu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, RIACS, 2001.

[83] U. Sammapun and O. Sokolsky. Regular expressions for run-time verification. In *Proceedings of the 1st International Workshop on Automated Technology for Ver ification and Analysis (ATVA'03)*, Taipei, Taiwan, December 10-12 2003.

[84] M. Schenke and E.-R. Olderog. Transformational design of real-time systems part i: From requirements to program specifications. *Acta Inf.*, 36(1):1–65, 1999.

[85] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91:100–111, 2003.

[86] V. Stolz and E. Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[87] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* PhD thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000.

[88] N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, New York, NY, USA, 2002. ACM Press.

[89] Úlfar Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM.

[90] Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[91] K. Zee, V. Kuncak, M. Taylor, and M. Rinard. Runtime checking for program verification. In *RV'07*, 2007.