RitHM: A Modular Software Framework for Runtime Monitoring Supporting Complete and Lossy Traces

by

Yogi Joshi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Runtime verification (RV) is an effective and automated method for specification based offline testing as well as online monitoring of complex real-world systems. Firstly, a software framework for RV needs to exhibit certain design features to support usability, modifiability and efficiency. While usability and modifiability are important for providing support for expressive logical formalisms, efficiency is required to reduce the extra overhead at run time. Secondly, most existing techniques assume the existence of a complete execution trace for RV. However, real-world systems often produce incomplete execution traces due to reasons such as network issues, logging failures, etc. A few verification techniques have recently emerged for performing verification of incomplete execution traces. While some of these techniques sacrifice soundness, others are too restrictive in their tolerance for incompleteness.

For addressing the first problem, we introduce RitHM, a comprehensive framework, which enables development and integration of efficient verification techniques. RitHM's design takes into account various state-of-the-art techniques that are developed to optimize RV w.r.t. the efficiency of monitors and expressivity of logical formalisms. RitHM's design supports modifiability by allowing a reuse of efficient monitoring algorithms in the form of plugins, which can utilize heterogeneous back-ends. RitHM also supports extensions of logical formalisms through logic plugins. It also facilitates the interoperability between implementations of monitoring algorithms, and this feature allows utilizing different efficient algorithms for monitoring different sub-parts of a specification.

We evaluate RitHM's architecture and architectures of a few more tools using architecture trade-off analysis (ATAM) method. We also report empirical results, where RitHM is used for monitoring real-world systems. The results underscore the importance of various design features of RitHM.

For addressing the second problem, we identify a fragment of LTL specifications, which can be soundly monitored in the presence of transient loss events in an execution trace. We present an offline algorithm, which identifies whether an LTL formula is monitorable in a presence of a transient loss of events and constructs a loss-tolerant monitor depending upon the monitorability of the formula.

Our experimental results demonstrate that our method increases the applicability of RV for monitoring various real-world applications, which produce lossy traces. The extra overhead caused by our constructed monitors is minimal as demonstrated by application of our method on commonly used patterns of LTL formulas.

Acknowledgements

I am highly obliged by the guidance and immense support given to me by Prof. Sebastian Fischmeister and Prof. Borzoo Bonakdarpour. I am highly grateful to them for providing me the opportunity to pursue research, and the skills, which they inculcated in me, would support me in future in my professional and personal life. Further, I am highly thankful to my thesis readers: Prof. Joanne Atlee and Prof. Richard Trefler.

I would also like to thank Dr. Guy Martin Tchamgoue, Ms. Greta Cutulenco, Ramy Medhat, Shay Berkovich, Wallace Wu, Dr. Samaneh Navabpour, Dr. Carlos Moreno, Dr. Thomas Reidmester, Yassir Rizwan, Dr. Akramul Azim, Jean-Christophe Petkovich, Ms. Paula Roser and Ms. Margaret Towell for their invaluable help and support.

Further, I thank University of Waterloo for honoring me with this opportunity to 'Learn to learn'. Last but not the least, I am immensely grateful to Prof. Stephen Mann for his invaluable guidance.

Dedication

This is dedicated to Prof. R. D. Ranade - my spiritual teacher, Prof. Sneha Joshi - my mother, Prof. Ramdas Joshi - my father, Mr. Hrushikesh R. Joshi - my brother, Late. Mr. Shridhar Joshi - my grandfather, and Mrs. Indumati Joshi, my grandmother.

Without them being there in my life, this would have never been possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview of Runtime Verification

Activities of software verification and testing constitute one of the most expensive parts of software development life cycle (SDLC), and their cost accounts for 50 to 75 percent of the total cost [36]. Issues caused by bugs and failures in the software can further create a huge financial impact for the software industry. Such loss has affected various application domains where software components are used. Recently National Institute of Standards and Technology (NIST) [75] found that around $59.6 billion are lost every year because of the issues in software systems. Bugs in financial softwares can directly cause monetary losses as in the case of a $3.45 billion over-payement of tax-credit made by Inland Revenue (from the United Kingdom) in 2004-2005 [19].

Furthermore, such loss is not limited to monetary considerations, but a loss of human lives can also occur due to software bugs. Modern embedded systems such as aircrafts and motor vehicles contain many software components, and the complexity of such software components in terms of size and functionality has grown recently. The avionics system of Airbus A380 contain more than 100 million lines of code [77]. The earlier model, Airbus A330, contains around 20 million lines of code. This shows that the complexity of software components in safety-critical embedded software is growing at a *highly* increasing rate. Consequently, this has increased the effort for testing to ensure the correctness of software systems. Any failures in such systems due to bugs can be fatal for the people, and such situations may result in loss of human-lives in addition to the financial losses. Thus, there has been an evolution of various techniques for effective verification of software for ensuring its correctness.

To ensure the correctness of software systems, various techniques such as formal verification and testing have been developed. Formal verification techniques such as model checking suffer due to the problem of state space explosion [21] as they require construction of exhaustive models of systems under verification [21]. Further, testing usually requires *exhaustively* covering all relevant execution paths and parts of the code being tested, and it is often infeasible to obtain such high coverage often due to the *increased* complexity of real-world systems. This often results in some execution paths of a code not being tested. With the growing complexity of software systems, it is highly important to provide *scalable* solutions to the problem of ensuring the correctness of such systems.

Runtime Verification (RV) [12, 39] is an *automated* technique where a *monitor* checks at run time whether or not the execution trace of a *program* satisfies some correctness properties. Depending on the observation of the monitor, the program can be *steered* into a correct state in case of a violation of the correctness criteria. Such *online* verification technique is highly useful for safety-critical embedded systems as their behavior can be validated and corrected at run time. Furthermore, this technique is useful for *offline* testing of software systems across multiple domains. Leucker et al. have depicted a taxonomy, which shows a variety of contexts in which RV can be used to ascertain the correctness of software systems [52]. This taxonomy highlights the scalability of RV techniques in various application domains.

A general setup of an RV framework is as shown in Figure 1.1. The details of various components of this setup are as follows. An observer extracts the current-state of a program, and a stream of such states forms an execution trace. Such execution trace, which is produced by an observer, is consumed by a monitor, which processes the trace to verify whether it satisfies the specified correctness criteria. Such correctness criteria is described using a specification language. A monitor-generator synthesizes a set of monitors from a set of high-level specifications, i.e., the monitor-generator translates the high-level specifications into low-level monitoring procedures. A monitor's output can be consumed by the program, and different steering actions can be performed to ensure that the program conforms to the correctness criteria.

We note that a general RV framework shares many features with compiler frameworks and database systems. The process in which a compiler translates a high-level code into low-level code is analogous to the process of synthesizing monitors from high-level specifications in a RV framework. This process is similar to the query compilation phase of a database infrastructure. An RV framework involves an additional process of running the generated monitors in an efficient manner. Although this process of running a set of monitors is analogous to that of query execution in a database system, the execution of RV monitors poses some unique challenges because of the extra overhead introduced by

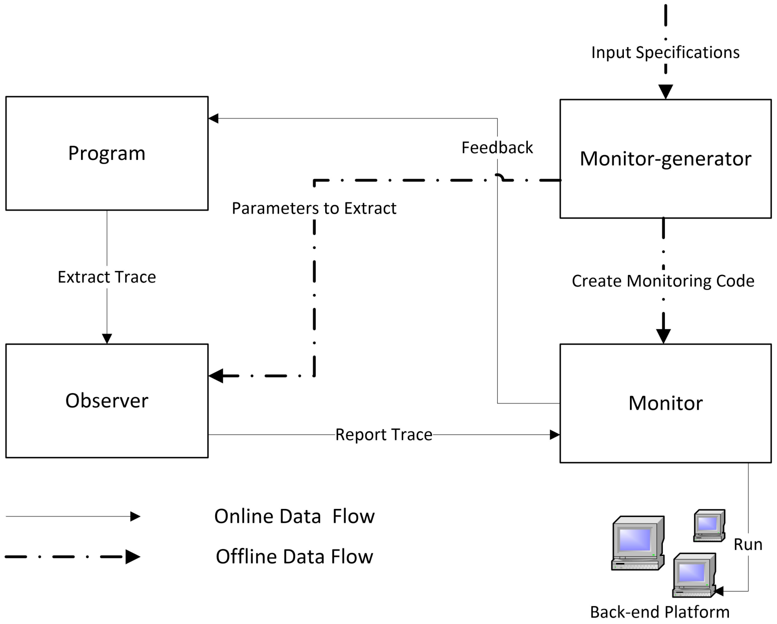


Figure 1.1: General RV Framework

the monitors, which run alongside the systems under verification.

Various compiler frameworks such as LLVM [51] have been successful due to their seminal role in the evolution of software development methods. A development of various high-level languages using compiler frameworks have enabled the developers to focus on domain aspects. The automation of low-level code generation has resulted in the reduction of defects and efforts during the process of software development. The development of various high-level languages using the compiler frameworks has resulted in huge savings in terms of the time and the cost of various software engineering tasks. Further, the automated query compilation techniques of database systems have resulted in huge savings in terms of money and time for the tasks of large-scale data management. The advanced techniques of query execution have enabled efficient processing of large-amounts of data.

We note that various proven design features of compiler frameworks and database systems can be adopted in an RV framework. A generic design of an RV framework (as shown in Figure 1.1) exhibits the isolation of the monitor generation process from the monitor execution process. The monitor generation process is generally run *offline*, i.e., before the execution of a program. On the other hand, the monitor execution happens alongside the execution of the program. Isolation of these components in the design of an RV setup enables independent optimizations of these tasks. The monitor-generator incorporates the features for facilitating the enhancement of specification languages, which we also refer to as logical formalisms. An RV framework needs support of different logical formalisms for describing specifications. A variety of logical formalisms, which differ in their expressivity are being used. Further, a single formalism cannot serve for different use-cases. Additionally, it is important to support multiple logical formalisms to allow a trade-off between the expressivity and efficiency because less expressive logical formalisms can often be monitored in a more efficient manner than more expressive ones.

The monitor execution process requires optimizations for efficiency in terms of invocation and execution overhead of monitors. Such optimizations include the ability of executing monitors on different back-end platforms such as multi-core CPU, GPU, FPGAs, etc. Further, the optimizations for efficient invocation schemes for monitors include buffer-triggered runtime verifiction (BTRV) [57], time- triggered runtime verification (TTRV) [17], etc. To reduce the overhead of monitors in terms of execution time and other resources, different efficient monitoring algorithms are utilized in the monitor execution process.

To enable a reuse of efficient monitoring algorithms, the interoperability between the different monitoring algorithms is important. Such interoperability between the monitors allows decomposing a specification into sub-specifications, and such sub-specifications can be monitored using efficient monitoring algorithms for respective fragments of temporal logic formalisms. Similar feature is designed in database systems, where a complex query is decomposed into different sub-queries, which are executed using efficient algorithms.

1.2 RitHM - A Comprehensive Framework

We surveyed various state-of-the-art RV frameworks and tools with respect to the aforementioned design considerations. The survey highlights a need for the development of a comprehensive framework for runtime verification. To facilitate a development of *scalable* RV techniques, we address the problem of identifying the important design features of an RV framework. With this motivation, we introduce RitHM, a comprehensive framework for RV. RitHM's architecture takes into account several important design considerations for increasing the effectiveness of an RV set-up in terms of following quality attributes: *usability*, *performance*, *modifiability* and *portability.* In general, achieving multiple quality attributes in a simultaneous manner involves trade-offs. Most existing RV tools and frameworks support only a few of the aforementioned quality attributes. RitHM's architecture highlights different trade-offs between these quality attributes along with respective design decisions.

Various design features of RitHM's architecture take into account state-of-the-art research on runtime verification. RitHM's features include support for multiple front-end and back-end plugins to enable *performance optimizations* for monitor execution and *extensions of logical formalisms* at the front-end. In the context of RV, a front-end is used for describing a set of specifications using logical formalisms, and the back-end performs a check to ascertain whether an execution trace satisfies the described specifications. Additionally, RitHM supports heterogeneous back-ends to take advantage of parallel algorithms for RV. RitHM also enables *interoperability* between different instances of monitors to leverage efficient monitoring algorithms for different fragments of logical formalisms. RitHM supports different types of plugins and flexible interfaces to facilitate integration into different software systems. Further, we report the results of experiments, which demonstrate the importance of various design features of RitHM.

Various advantages of RitHM's design features are demonstrated via architecture evaluation using architecture trade-off analysis method (ATAM) [48]. We also provide evaluations of a few other tools using ATAM. These evaluations highlight the importance of various unique design features of RitHM. To our knowledge, this is one of the first architectural evaluations of a framework in the RV community. We also report the applicability of RitHM's features through case-studies where real-world applications are monitored using RitHM.

1.3 Runtime Verification of Lossy Traces

RV imposes additional overhead at run time partly due to the techniques upon which it relies on to extract the traces from the monitored programs. For this purpose, many techniques have been developed to reduce the overhead of monitoring tools. Sampling-based techniques [4, 32, 38, 44] periodically extract events from a monitored program.

All these techniques generally trade-off information and overhead, and may produce incomplete traces as some events may not be observed. However, many other reasons may cause a system to produce execution traces with a loss of events. For example, if an execution trace is being sent over a network to a remote *monitor*, a network failure may cause loss or even corrupts some events in the trace. Thus, most network protocols such as transmission control protocol (TCP) [23] and trivial file transfer protocol [73] incorporate various loss tolerance techniques such as automatic repeat request (ARQ) [30] and forward error correction (FEC) [45]. Data loss is also prevalent in sensor networks and other wireless communication systems due to noise, collision, unreliable link, weak signal, and unexpected damages [50, 66]. When not sent over network, a logging failure [7] can also cause events to be lost. Even a sound monitor, designed for complete traces, may end up delivering an incorrect verdict for a given specification when it processes an incomplete execution trace . It is therefore important to develop runtime monitors that yield sound verdict even in presence of an execution trace with a loss of events.

While most RV techniques assume the existence of complete traces [14, 41, 42, 43], a few approaches have recently been developed for traces with a loss of sequence of events. Stoller et al. [74] proposed the concept of runtime verification with state estimation (RVSE), where the probability of whether a specification is satisfied or violated by a program is calculated by extending the classic forward algorithm for Hidden Markov Model (HMM). However, this approach is limited by the requirement of a comprehensive set of execution traces used to learn the corresponding HMM. Thus, if there is no execution trace that violates the specification, then a monitor when used along with corresponding learned HMM, may deliver an unsound verdict for a specification. Basin et al. [7] showed that not all logging failures can affect the truth-values of the formulas in a particular fragment of metric first-order temporal logic (MFOTL). As per their approach, if the truth-value of a formula cannot be determined due to logging failures, then the three-valued semantics are used to express the uncertainty about the verdict. Further, if lost events of a trace are recovered, then such uncertainty can be resolved. Consequently, this approach relies on run time conditions, which are determined by execution traces, and therefore lacks a clear classification of properties depending upon their monitorability on incomplete traces.

We present a novel approach for the problem of runtime verification of LTL (Linear Temporal Logic) [65] formulas on execution traces with loss of a sequence of events. We identify a fragment of LTL specifications which can be *soundly* monitored in presence of *transient* loss of a sequence of events in an execution trace. Evaluation of LTL formulas can be sensitive to loss of events. A finite loss of events results in a failure of a monitor to observe and process the corresponding events. Consequently, such a monitor may deliver an unsound verdict for the monitored LTL formulas, which can be either violated or satisfied over a finite execution trace. For example, a *guarantee* [31] formula which says that a thread should *eventually* execute the system call *pthread_exit* can be expressed in LTL as $\Diamond\, thread\_exit$. If a monitor does not observe the event when the thread calls *pthread_exit*, then it would deliver an incorrect verdict saying that the specification is not satisfied by the program. Similar arguments can be provided for *safety* [31] and *obligation* [31] formulas.

On the other hand, some LTL formulas can never be satisfied or violated over finite traces. For example, an LTL formula $\Box\Diamond\,(garbage\_collector\_invoked)$ captures the idea that a garbage collector should be invoked infinitely often. A monitor for this formula can tolerate a finite loss of a sequence of events. Intuitively, even if finitely many events about the invocation of the garbage collector are not observed, the monitor can still determine the verdict of this LTL formula based upon the observation of subsequent events. When a garbage collector is invoked, it is guaranteed that it collects the garbage accumulated in the past where the execution trace was not observed. Provided that the loss of a sequence of events is transient, i.e., the monitor can eventually process the next stream of events from the execution trace after a finite loss, the LTL formula can be verified successfully on the execution trace in spite of the finite loss of a sequence of events. Although such formulas can never be satisfied or violated by finite traces, Falcone et al. [31] have shown that they are *monitorable*. Moreover, such formulas belong to the commonly used patterns [29] of LTL formulas. A study by Bauer et al. [14] shows that many of the commonly used LTL formulas can never be satisfied or violated by finite traces.

Falcone et al. [31] proposed that if a monitor's verdict for a good (finite) trace differs from its verdict for a bad (finite) trace, then the corresponding formula is deemed as monitorable because the verdict can then be used to execute different steering actions. The described approach uses the four-valued semantics for LTL. For example, consider an LTL formula $\Box(job\_killed \rightarrow \Diamond job\_resubmitted)$. If a finite trace ends when only job_killed is $\top$, then the verdict of the monitor for this trace is $\bot_p$, i.e., 'presumably false'. On the other hand, if a finite trace ends when $job\_resubmitted$ is $\top$, then the verdict for the monitor for this trace is $\top_p$, i.e., 'presumably true'. Such verdicts then can be used to take further recovery actions for the job.

Building on these results, we show that some LTL properties can indeed be monitored in spite of a finite loss of events in an execution trace. We describe an algorithm, which finds whether an LTL formula can be successfully monitored in case of a *transient* loss of events. Since a transient loss is not permanent, it is ensured that a system that encounters such a loss *eventually* recovers from the loss to observe subsequent events. Thus, after a transient loss of events, the monitor also eventually processes the next stream of events. This consideration holds for most real-world systems, in which recovery mechanisms are implemented to ensure that a loss is not persistent. Furthermore, we extend the monitor construction method of Bauer et al. [14] to synthesize monitors for monitorable formulas in presence of a transient loss of a sequence of events in an execution trace. Additionally, it is expected that a monitor that process an incomplete execution trace should produce a verdict, which aligns with that of a monitor that processes the corresponding complete trace. The verdict should then remain the same even when the lost events in the trace are recovered. To capture this notion, we introduce the concept of *monotonicity*. Thus, a monitor processing an incomplete trace is monotonic iff its verdict always matches the verdict of a monitor which processes the corresponding complete execution trace irrespective of whether the events lost in the past are recovered. We note that the probabilistic monitoring approaches [70, 74] do not exhibit this behavior as the verdict of their monitors may change in case lost events are recovered.

We further describe an example of a similar formula, which depicts a typical scenario during usage of the commonly used network protocols. The LTL property is $\Box(\mathit{request} \rightarrow \Diamond\, \mathit{response})$. Informal description of this property states that it is always a case that a *request* is eventually followed by a *response*. This property can be monitored in spite of a finite loss of events in the execution trace provided that the monitor observes at least one *request* or *response* action in the system after the finite loss of events. The observation of at least one *request* or *response* in the subsequent stream of events ensures that the monitor produces a sound verdict in spite of not being able to process a finite number of events in the execution trace. Thus, the LTL formula is monitorable in spite of a finite loss of events provided certain conditions are met by the execution trace.

1.4 Contributions

The contributions[1] of this thesis can be summarized as follows.

Design Considerations for RV frameworks: We analyze various state-of-the-art RV tools and frameworks along with various monitoring algorithms. Based on this survey, we identify various important design features of RV frameworks. We note that modifiability, efficiency, usability and portability are the important quality attributes of an RV setup.

A Design of RitHM Framework: Based upon the identification of the quality attributes, we develop a design for RitHM, a comprehensive framework for runtime verification. We perform various experiments to validate the feasibility and importance of the proposed design features. Based upon the evidences, we provide an implementation of RitHM framework.

An Evaluation of RitHM's Architecture: In addition to the empirical evaluation, we performed evaluations of RitHM's architecture using architecture trade-off analysis method (ATAM) during the design phase. Additionally, we evaluated few other tool's architecture using ATAM. This is one of the first architectural evaluations frameworks in the RV community. We note sensitivity and trade-off points of the architectures during our evaluation.

[1] Work presented in this thesis is submitted (or is under preparation for submission) in conference papers and journals.

Empirical Evaluation of RitHM: We demonstrate the applicability and importance of various design features of RitHM using a few case-studies. We perform monitoring of Google cluster traces and the traces of an Engine Control Unit (ECU) of a Toyota 2JZ engine. A use of interoperability of parallel and sequential monitoring algorithms demonstrates a significant gain in terms of performance during the monitoring of Google cluster traces. Moreover, the importance of *usability* feature of RitHM's front-end components is demonstrated by a use of rewriting feature, where high-level specifications for the ECU traces are rewritten into corresponding low-level specifications. Thus, users of RitHM can use this feature to abstract the complexity of describing specifications in low-level specification languages.

Identifying Monitorability Criteria for Loss-tolerant Monitoring: As a second part of the thesis, we describe the importance of the problem of runtime verification of lossy execution traces, which often result due to network failures, partial instrumentation, logging failures, etc. Further, we describe a model, where execution traces exhibit a *transient* loss of events. Under this model, we identify the monitorability criteria for LTL formulas.

Constructing Loss-tolerant LTL Monitor: We present an *offline* algorithm to construct loss-tolerant monitors for LTL formulas. We evaluate the complexity of our algorithm by applying it on commonly used patterns of LTL formulas. This application shows the effectiveness of the constructed monitors in terms of memory usage. We also prove the correctness of the monitors constructed using our algorithm.

Monotonicity: We define the concept of monotonicity of a monitor's verdict under a transient loss of events. This notion evaluates loss-tolerant monitors based upon a possibility of the invalidation of their verdicts in the event of a recovery of lost events in traces. In other words, the verdict of a monitor that processes an incomplete trace should match that of a monitor that processes the corresponding complete execution trace, irrespective of whether the lost events are recovered. We prove that the monitors constructed using our method provide a correct verdict irrespective of whether a recovery of the lost events is possible. We compare other approaches w.r.t. the notion of monotonicity and report our findings.

Empirical Evaluation for Loss-tolerant Monitoring: We describe the applicability of our approach by two case studies. Our results show that our approach increases the applicability of runtime verification for real-world applications, which often produce lossy traces. Additionally, our approach can be used to control the monitoring overhead for some quality-of-service-aware software applications such as MPlayer.

1.5 Organization

The organization of this thesis is as follows. In Chapter 2, we present the details about related work about various tools for runtime verification and the related discussion. We also present the background information for our work on runtime verification in presence of transient loss of events in an execution trace.

In Chapter 3, we present design and architecture of RitHM. In Chapter 4, we provide the details of various design considerations for RitHM along with different experiments, which confirm our claims about the design features. Furthermore, we report the results of evaluation of architectures using ATAM. In Chapter 5, we provide details about various case studies in which RitHM is used to monitor real-world datasets. Further, we discuss the details about the lessons we learned during the implementation of RitHM.

In Chapter 6, we discuss the formal problem description and the criteria for monitorability of an Ltl formula in presence of transient loss of events in an execution trace. In Chapter 7, we provide a high level description of our solution. Further, we present an algorithm for verifying the monitorability and synthesizing a loss-tolerant monitor for Ltl formulas. We also provide proofs related to correctness for monitors generated using our algorithm. In Chapter 8, we present case studies, which depict the evaluation of the proposed method on real-world applications and datasets. Finally, Chapter 9 describes the details of further extensions of our work along with our concluding remarks.

Chapter 2

Background and Related Work

2.1 Overview of Various RV tools and techniques

Since the advent of RV, various tools have been developed to address the problem of runtime verification for various application domains. In this section, we provide a brief overview of several tools. Some of these tools such as Tracematches [1] and J-LO [16] use aspect-oriented programming for extracting the runtime state information from a running program. These tools can monitor *formal properties* written in specific formal languages such as linear temporal logic (LTL) for J-LO and *parametric* regular patterns for Tracematches. jUnitRV [26] is a tool, which supports verification of Ltl formulas within JUnit test framework. jUnitRV uses RV-LTL [13] semantics, which is one of the finite-path semantics for Ltl. jUnitRV is primarily designed for monitoring in a test setup. RiTHM [63] uses Ltl as a specification language. Further, it uses time-triggered runtime verification (TTRV) for reducing the overhead. RiTHM can be used for *online* monitoring during the execution of a program as well as for *offline* monitoring of traces produced by program(s). Further, RiTHM implements parallel algorithms [15] using GPU for reducing the overhead of verification. LOLA [24] is a tool, which has developed its own specification language. This language allows it to encode Ltl semantics using recursive expressions. LOLA allows numerical constraints over various variables in specifications. Further, a LOLA specification consists of multiple expressions, whose output can be used as an input to other expressions.

MARQ [67] uses Quantified Event Automata (QEA) [6], an expressive formalism, to express *parametric* temporal properties, and the resultant monitors can be integrated in Java programs for online monitoring using AspectJ. MonPoly [8] uses a safety fragment

Tool	**Logical Formalism(s)**	**Back-ends for Monitor Execution**	**Supports Interoperability?**
J-LO [16]	Parametric LTL	CPU	No
LOLA [24]	LOLA Language	CPU	No
MONPOLY [8]	MFOTL	CPU, Multiprocessor	No
jUnitRV [26]	LTL	CPU	No
MARQ [67]	QEA	CPU	No
MOP [58]	Multiple*	Multiple*	No
BeepBeep [37]	FOLTL	CPU	No
RiTHM [63]	LTL	CPU, GPU	No

Table 2.1: Features of RV Tools

of metric first-order temporal logic (MFOTL), which allows aggregation operators such as *SUM, COUNT, MIN,* etc. Most of these tools are designed for specific formal languages, which we also refer to as *logical formalisms*. MOP (Monitoring Oriented Programming) [58] is a framework for RV, and it allows different logic plugins to allow monitoring of software specifications specified in different logical formalisms. JavaMOP, which is an instance of MOP framework, is used for online monitoring of Java programs using AspectJ. BeepBeep [37] is a tool for performing RV of web applications. It uses first-order linear temporal logic (FOLTL) for describing specifications. Breach [27] is a toolbox for verification of hybrid systems, and it supports various logical formalisms such as MITL (Metric Interval Temporal Logic) and STL (Signal Temporal Logic).

We present the summary of these tools in Table 2.1. As shown, most of the tools and frameworks are designed for monitoring specifications described using a specific logical formalism. Only MOP framework allows logic plugins along with potential support for multiple back-ends for efficient execution of monitoring algorithms. Further, none of the tools support the interoperability feature, which allows to decompose complex specifications into multiple sub-specifications, and these sub-specifications could be monitored separately using efficient algorithms.

In addition to various tools, several efficient algorithms [6, 9, 10, 14, 56, 76] have been developed for logical formalisms such as LTL, MFOTL, QEA, FOLTL, regular expressions, context-free grammars and various fragments of these formalisms. These algorithms differ in their time and space complexities. Further, these algorithms often address the problem of verification for a particular fragment of a logical formalism. For example, in [56], Medhat et al. have developed an efficient parallel algorithm for a fragment of first-order LTL. This algorithm requires that the FOLTL formula conforms to certain syntactical

Logical Formalism	Algorithm	Fragment Only?	Heterogeneous Back-ends?
LTL	LTL_3 based State Machine [14]	✔	✗
	RV-LTL based State Machine [13]	✗	✗
	GPU based Algorithm [15]	✗	✔
	Map-reduce based Algorithm [5]	✗	✔
MTL	Offline Algorithm based on [8]	✔	✗
	Recursion Based Online Algorithm [35]	✔	✗
first-order LTL	Sequential Algorithm (based on LTL_3 and RV-LTL)	✗	✗ ✗
	Parallel Algorithm for multi-core and GPU [56]	✔	✔
MFOTL	Offline Algorithm (based on MTL Algorithm in [8])	✔	✗
	Recursive definition based Algorithm	✔	✗
	Parallel Algorithm for Multi-core (Adapted from first-order LTL algorithm [56])	✔	✔
QEA	QEA Monitor [67]	✗	✗

Table 2.2: Algorithms for Widely Used Logical Formalisms

restrictions. Further, in [35], Gunadi et al. have developed an efficient algorithm using recursive definitions for a fragment of past time metric first-order temporal logic. The syntactic restrictions on this fragment allow a use of recursive definitions of timed 'Since' operator similar to that of past time linear temporal logic (PTLTL) [41]. Barre et al. [5] have developed a map-reduce based algorithm for efficient verification of LTL.

Table 2.2 shows a summary of various logical formalisms and available algorithms, which are implemented by aforementioned tools. Column 3 of the table indicates whether the algorithm only applies to a fragment of the respective logical formalism. Column 4 of the table indicates whether the algorithm can be run on heterogeneous back-ends such as multi-core CPU, GPU, etc.

An RV framework uses the state information of a running program for performing checks. This state information is obtained in multiple ways. There are a variety of *static* and *dynamic* instrumentation tools such as DIME [3], Pin [54], AspectJ, Java Management Extensions (JMX [46]) based utilities, LLVM based utilities [51], LTTng [53], etc. These tools implement different optimizations for efficient instrumentation. Moreover, these tools facilitate instrumentation in various established formats. Further, there are different in-house tools used in the software development industry for the purpose of performing instrumentation. Among the aforementioned RV tools, J-LO, Tracematches, QEA and JavaMOP use AspectJ for instrumentation. MonPoly has been primarily used for offline monitoring of *recorded* log files. jUnitRV uses Javaassist [20] to inject code at load-time of the Java classes of a running Java program. RiTHM uses LLVM based utility for instrumentation. A use of JMX is a widely adapted method for monitoring of Java EE applications.

2.2 Competition on Software for Runtime Verification (CSRV'14)

CSRV'14 was held as a initiative to compare various tools on the basis of their efficiency in terms of monitoring overhead and their expressiveness. Further, another motivation behind this competition was to come up with a set of benchmarks that can be used in the future to assist the performance evaluation of various RV tools. RiTHM [63], a tool from our previous work, was a part of this competition (see `http://rv2014.imag.fr/monitoring-competition/results` for more details).

CSRV'14 consisted of three phases for each of its tracks. In the first phase, every team supplied a set of benchmarks using the specifications with logical formalisms and respective data-sets or programs to be monitored. In the next phase, every team would *tune* their tool for the benchmarks of all other teams in the respective track. The tuning phase involved rewriting the specifications from other teams into a supported logical formalism, and if required, existing logical formalisms were *extended* to express different specifications. Further, this phase involved tuning different monitoring algorithms developed by various research groups for their logical formalisms. Finally, in the last phase, score was calculated for each team based on the *monitoring overhead* for each benchmark and correctness of the verdicts of the monitors. The overhead was assessed in terms of *memory* and *CPU consumption* of a monitor in addition to its latency in delivering a verdict.

2.3 Discussion

Different approaches for development of aforementioned tools and the respective research work show that *expressiveness* of logical formalisms and *efficiency* of runtime monitoring are among the most important factors under consideration. While more expressive logical formalisms allow specifying complex properties, efficient techniques for monitoring reduce the overhead at runtime. A typical RV setup consists of a front-end, which allows specifying the properties using logical formalisms, and the back-end consists of the components, such as instrumentation tools and monitors, which perform a check of whether an execution trace satisfies the described specification. Typically, the back-end is optimized for performance as it runs alongside the program under verification. Further, it utilizes overhead control techniques such as parallel monitoring algorithms, TTRV, etc. On the other hand, the front-end is designed for processing a set of specifications. Further, Table 2.2 shows the availability of different monitoring algorithms for various fragments of a logical formalism. Hence, a segregation of front-end and back-end of an RV setup can enable *reuse* of a front-end for different back-ends, and vice versa.

Further, we note the fact that many of the logical formalisms that are used by aforementioned tools are interrelated. Various operators used by different formalisms have identical semantics. For example, MonPoly [8] tool uses aggregation operators such as *SUM*, *COUNT*, whose semantics are the extensions of the semantics of quantifiers used in first-order logic. For a fragment of first-order LTL specifications [56], Medhat et al. have described an algorithm, where a set of RV-LTL [13] monitors are created to monitor the first-order Ltl formula. These observations show a trend in the evolution of logical formalisms, where more complex and expressive formalisms get evolved from the less complex ones. In other words, different temporal logics have evolved by extension or augmentation of the semantics of other logical formalisms. As previously discussed, different algorithms have been developed for efficiently monitoring the respective fragments of logical formalisms, and these algorithms can be reused for monitoring of further extensions of the respective logical formalisms. Additionally, a software system may exhibit specifications, which require extending the semantics of an existing logical formalism. Here, we consider an example of a safety specification for soft real-time systems. Informally, the specification states that for 90 percent of the *threads*, it is *always* the case that if a thread is *ready*, then it *eventually* gets scheduled to run *within* x time units. This property can be captured by a variant of MFOTL as $\Box\,(\mathit{Percentage}(t) \cdot \Box(\mathit{ready}(t) \;\rightarrow \Diamond[0,x]\,\mathit{running}(t)) \geq 90)$. The semantics of aggregation function 'percentage' are similar to the semantics of quantifiers $\forall, \exists$. Hence, an existing implementation of a MFOTL monitor can be extended to monitor the specifications with such new aggregation operators.

Furthermore, verification of diverse types of software systems demand writing the specifications using different variants of the similar logical formalisms. For example, a MTL specification may consists of different sub-formulas, where one of the sub-formulas conforms to a particular syntax for which an efficient parallel-algorithm is available. In this case, different monitors can be *spawned* for different sub-formulas, and such monitors can run in parallel. The verdicts of such monitors could be *combined* to arrive at the final verdict. Such optimization can prove more efficient than monitoring the entire formula using a generic monitoring algorithm. Hence, a support for *interoperability* between monitors is an important aspect for providing efficient ways of runtime monitoring. Figure 2.2 shows the availability of multiple efficient algorithms for fragments of various logical formalisms, and a support for interoperability can allow integrating the features of such algorithms for efficient monitoring of different formulas.

To facilitate an application of RV tools in industry, it is important that higher level abstractions of specification languages become available. In general, programmers are reluctant to write formal specifications because of the complexity involved in understanding the formalisms [2]. Thus, as argued by Ammmons et al., verification techniques are unlikely to be widely adopted unless cheaper and easier ways of formulating specifications are developed. High-level abstractions of current logical formalisms can make it easy for the practitioners to create and monitor specifications with minimal assistance from experts.

Further, our experience of CSRV'14 confirmed our analysis that logical formalisms, which are at lower level of abstraction, can express specifications written in other higher level logical formalisms. MARQ [67], which uses QEA as the logical formalism, was able to express specifications in most other logical formalisms. While such encoding is useful, high-level abstractions are easy to understand for users. Thus, it is important to segregate the specification processing phase from the monitor construction phase of an RV setup. The specification processing phase would need to support translation of a specification, written using a high-level specification language, into an equivalent specification in a low-level language for which efficient monitor construction algorithms already exist.

The back-end component of an RV setup requires efficient monitoring algorithms for controlling the overhead. Instrumentation and monitor execution are the activities, which mainly contribute for this exta-overhead. There have been different schemes developed to counter this overhead [4, 17, 32, 38, 44]. As discussed previously, there are different parallel monitoring algorithms, which utilize GPU, multi-core CPU, FPGAs, etc. Such techniques require an RV setup to support multiple back-ends, which enable efficient runtime monitoring. It implies that there needs to be a segregation of the monitor component from other components of a typical RV setup. As shown in Figure 2.2, some monitoring algorithms use multi-core CPU and GPU for performing efficient runtime monitoring.

With reference to aforementioned considerations, we note that available state-of-the-art tools and frameworks for runtime verification do not offer direct support for extending logical formalisms and reusing various efficient monitoring algorithms. Further, many software projects often develop proprietary test beds, and an RV framework, with a comprehensive set of APIs, can help automation developers to integrate state-of-the-art monitoring algorithms into their software applications and test beds. Further, efficient monitoring algorithms with different back-ends and overhead-control schemes can help to reduce the exta-overhead caused by RV. Incorporating the interoperability feature between a set of monitors adds another dimension to reducing the overhead of RV, where efficient algorithms for different fragments of a logical formalism can be utilized.

2.4 Preliminaries on Runtime Verification of LTL

In this section, we provide an overview of LTL along with the details of its finite path semantics.

2.4.1 Overview of LTL

A program P is considered to be a generator of *computation*, i.e., an infinite sequence of *events* or *states*. We use the symbol Σ to denote the set of states, which is also referred to as an *alphabet*. Thus, an infinite sequence of elements from an alphabet Σ is called an *infinite word*. Similarly, a finite sequence of elements from an alphabet is referred to as a *finite word*. Further, we use Σ^* and Σ^+ to designate the set of all finite words and the set of all nonempty finite words, respectively. Also, we use Σ^ω to refer to the set of all infinite words. A program P produces an execution trace σ, and a *monitor* $\mathcal{M}$ provides a verdict stating whether σ *satisfies* a specification φ. The concatenation of a finite trace σ and another trace ρ is denoted by $\sigma.\rho$. Finally, the i^{th} element of a word $\sigma = a_1, a_2, \ldots$ is referred to as a_i.

Definition 1 (Syntax of LTL). *Let AP be a finite and non-empty set of* atomic propositions*, and* $\Sigma = 2^{AP}$ *a finite* alphabet*. The set of* LTL *specifications is inductively defined as follows:*

$$\varphi ::= \ true \mid p \mid \neg\, \varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\, \varphi \mid \varphi_1\, \mathbf{U}\, \varphi_2$$

Where, $p \in AP$*, and* $\bigcirc$ *(next), and* $\mathbf{U}$ *(until) are the temporal operators. The boolean operators retain their usual meaning.*

Definition 2 (Semantics of LTL). *Let* $\sigma = a_1, a_2, \ldots$ *be an infinite trace in* Σ^ω*.* LTL *semantics are inductively defined over the infinite trace as per below:*

$$\begin{array}{lll}
\sigma, i \models \top & & \\
\sigma, i \models p & \text{iff} & p \in a_i \\
\sigma, i \models \neg\varphi & \text{iff} & \sigma, i \not\models \varphi \\
\sigma, i \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma, i \models \varphi_1 \ \vee \ \sigma, i \models \varphi_2 \\
\sigma, i \models \bigcirc\varphi & \text{iff} & \sigma, i+1 \models \varphi \\
\sigma, i \models \varphi_1\, \mathbf{U}\, \varphi_2 & \text{iff} & \exists k \geq i : \sigma, k \models \varphi_2 \wedge \\
 & & \forall j : i \leq j < k : \sigma, j \models \varphi_1
\end{array}$$

Where $\models$ *denotes the satisfaction relation.*

Further, $\sigma \models \varphi$ is *true* iff $\sigma, 1 \models \varphi$. We introduce syntactic sugar in the form of two operators $\Box$ (*always*) and $\Diamond$ (*eventually*). Thus, $\Diamond\varphi$ is defined as $\mathit{true}\,\mathbf{U}\,\varphi$, and $\Box\varphi$ is defined as $\neg\Diamond\neg\varphi$. An LTL formula φ defines a set of traces, which we refer to as $L(\varphi)$. A trace σ satisfies an LTL formula φ if $\sigma \in L(\varphi)$. Chang et al. [18] have classified LTL formulas into the following six classes: safety, guarantee, obligation, response, persistence, and reactivity.

2.4.2 Finite Path Semantics for LTL

LTL semantics are defined over infinite words. However, in practice, a program can only generate a finite word. Since the problem of runtime verification considers only finite words, multiple *finite path semantics* have been developed for LTL. Such semantics include LTL$_3$ [14], FLTL [55], RV-LTL [13], and asymptotically correct finite path semantics [61]. In this subsection, we provide a brief description of some of these semantics.

LTL$_3$ The semantics of LTL$_3$ was introduced by Bauer et al. [14]. LTL$_3$ uses three truth values to denote the evaluation of an LTL formula over finite paths. These three truth-values are from the truth-domain, $\mathbb{B}_3 = \{\top, ?, \bot\}$. Let $\models_\omega$ denote the satisfaction relation over infinite paths.

Definition 3 (LTL$_3$ semantics)**.** *Let $\sigma = a_1, a_2, \ldots \in \Sigma^*$ denote a finite trace. The evaluation of the truth-value of an LTL$_3$ formula φ for the trace σ is defined as follows:*

$$[\sigma \models_3 \varphi] = \begin{cases} \top & \text{if } \forall\, v \in \Sigma^\omega : \sigma.v \models_\omega \varphi \\ \bot & \text{if } \forall\, v \in \Sigma^\omega : \sigma.v \not\models_\omega \varphi \\ ? & \text{otherwise} \end{cases}$$

Bauer et al. [14] also proposed a monitor synthesis method to generate monitors for LTL formulas. A monitor has an output function $\lambda : \mathcal{Q}_i \rightarrow \mathbb{B}_3$, i.e., each state of the FSM is mapped to one of the truth-values in the truth-domain $\{\top, \bot, ?\}$. An LTL$_3$ monitor delivers a verdict as $\top$ on a finite word provided *all* the infinite extensions of the finite word satisfy the LTL formula. In this case, the finite word is called a *good* prefix of the LTL formula. On the other hand, an LTL$_3$ monitor delivers a verdict as $\bot$, when *all* the infinite extensions of the finite word violate the LTL formula. Otherwise, an LTL$_3$ monitor delivers a verdict as ?, i.e., the truth-value of the specification is *unknown.* Furthermore, a finite prefix σ is called an *ugly* prefix, when there exists no other *finite* prefix u such that $\sigma.u$ is either a *good* or a *bad* prefix.

An LTL formula is *monitorable* [14] provided there exists no ugly prefix for the formula.

FLTL [55] The semantics of LTL are defined over infinite traces. Finite LTL (FLTL) was proposed to reason about finite traces for verifying formulas at run time as a monitor can only observe a finite-word at run time. FLTL semantics are based on the truth values $\mathbb{B}_2 = \{\top, \bot\}$. FLTL definition contains a weak 'next' operator denoted by $\overline{\bigcirc}$.

Definition 4 (FLTL semantics)**.** *Let φ and ψ be two* LTL *formulas, and $\sigma = a_0 a_1 \cdots u_{n-1}$ be a finite trace. Also, let ϵ denotes an empty trace.*

$$[\sigma \models_{\mathrm{F}} \bigcirc \varphi] = \begin{cases} [a_1 \models_{\mathrm{F}} \varphi] & \text{if } a_1 \neq \epsilon \\ \bot & \text{otherwise} \end{cases}$$

$$[\sigma \models_{\mathrm{F}} \overline{\bigcirc} \varphi] = \begin{cases} [a_1 \models_{\mathrm{F}} \varphi] & \text{if } a_1 \neq \epsilon \\ \top & \text{otherwise} \end{cases}$$

$$[\sigma \models_{\mathrm{F}} \varphi \,\mathbf{U}\, \psi] = \begin{cases} \top & \text{if } \exists k \in [0, n-1] : [a_k \models_{\mathrm{F}} \psi] = \top \ \wedge \\ & \forall l \in [0, k) : [a_l \models_{\mathrm{F}} \varphi] = \top \\ \bot & \text{otherwise} \end{cases}$$

Although not all LTL formulas are monitorable with LTL_3 [14], such formulas belong to the commonly used patterns [14, 29] of LTL formulas. For example, a formula $\Box(a \rightarrow \Diamond b)$ is not monitorable with LTL_3 because it cannot be satisfied or violated by a finite word, and the corresponding LTL_3 monitor consists of a single state with its verdict as 'unknown'. To address this problem, Bauer et al. [13] combined FLTL semantics with LTL_3 semantics to derive RV-LTL semantics. Here, a weak 'next' operator evaluates to $\top$, when a next state does not exist at the end of a finite word. Further, the strong 'next' operator evaluates to $\bot$, when a next state does not exist at the end of a finite word. In order to combine FLTL with LTL_3, $\Diamond$ is evaluated using a strong 'next' operator, and $\Box$ is evaluated using a weak 'next' operator. This captures the idea that $\Diamond \varphi$ needs to be evaluated *negatively*, when φ has not been satisfied in any of the observed states. Further, $\Box \varphi$ needs to be evaluated *positively*, when φ is not violated in any of the observed states.

Definition 5 (RV-LTL semantics)**.** *Let $\sigma = a_1, a_2, \ldots \in \Sigma^*$ denote a finite trace. The evaluation of truth-value of a RV-LTL formula φ for the trace σ is denoted as $[\sigma \models \varphi]_{RV}$*

which is an element of $\mathbb{B}_4$ *and is defined as follows:*

$$[\sigma \models \varphi]_{RV} = \begin{cases} \top & \text{if } \forall\, v \in \Sigma^\omega : \sigma.v \models_\omega \varphi \\ \top & \text{if } \forall\, v \in \Sigma^\omega : \sigma.v \not\models_\omega \varphi \\ \top_P & \text{if } [\sigma \models \varphi]_3 = \text{?} \text{ and } [\sigma \models \varphi]_F = \top \\ \bot_P & \text{if } [\sigma \models \varphi]_3 = \text{?} \text{ and } [\sigma \models \varphi]_F = \bot \end{cases}$$

Further, Falcone et al. [31] described an alternative definition of *monitorability*. Based upon their definition, an Ltl formula is *monitorable* iff the constructed monitor can distinguish between *good* and *bad* finite prefixes, i.e.,

$$\forall\, \sigma_{good} \in L^*(\varphi) \cdot \forall\, \sigma_{bad} \in L^*(\neg\varphi) \cdot [\sigma_{good} \models \varphi]_{\mathbb{B}} \neq [\sigma_{bad} \models \varphi]_{\mathbb{B}}$$

Here, $\mathbb{B}$ denotes the truth-domain of the finite path semantics, used for the evaluation of the formula. $L^*(\varphi)$ denotes the set of *good* finite prefixes for an Ltl formula φ. Bauer et al [13] provide a method for constructing a monitor, as a deterministic FSM, for an RV-LTL formula. As shown in [13], RV-LTL satisfies the maxim of 'Complementation by negation'. Thus, RV-LTL monitors can be used to monitor the Ltl formulas, which can never be satisfied or violated by finite traces because RV-LTL monitors conform to the requirements of the alternative definition of monitorability by producing a different verdict for good and bad finite prefixes of words.

Definition 6 (RV-LTL Monitor)**.** *Let* φ *be a RV-LTL formula over an alphabet* Σ*. The* monitor $\mathcal{M}^\varphi$ *of* φ *is the FSM* $(\Sigma, Q, q_0, \delta, \lambda)$*, where* Q *is a set of states,* q_0 *is the initial state,* $\delta \subseteq Q \times \Sigma \times Q$ *is the transition relation, and* λ *is a function that maps each state in* Q *to a value in* $\{\top, \top_p, \bot_p, \bot\}$*, such that,*

$$[\sigma \models \varphi]_{RV} = \lambda(\delta(q_0, \sigma))$$

Chapter 3

RitHM Framework

With a motivation to further improve the current state of RV techniques w.r.t. our analysis of the state-of-the-art tools and techniques, we propose RitHM (Runtime Heterogeneous Monitoring), a comprehensive framework, whose architecture incorporates design features to achieve following goals. These goals focus on generating efficient monitors in addition to supporting expressive logical formalisms and their extensions.

3.1 Goals of RitHM's design

- Facilitate development of extensions of logical formalisms and their high level abstractions.
- Ability to run monitors on multiple back-ends with different overhead-control schemes in order to support efficient monitoring techniques.
- Support interoperability of monitors so that efficient algorithms for fragments of logical formalisms could be reused.
- Provision of APIs in order to facilitate easy integration with instrumentation tools, existing applications and testing frameworks.
- Provide a platform for comparative evaluation of different monitoring algorithms.

Besides being used for runtime verification, we believe that RitHM would serve as an open-source repository of interoperable implementations of cutting-edge monitoring algorithms developed for different logical formalisms. It would assist to drive further research

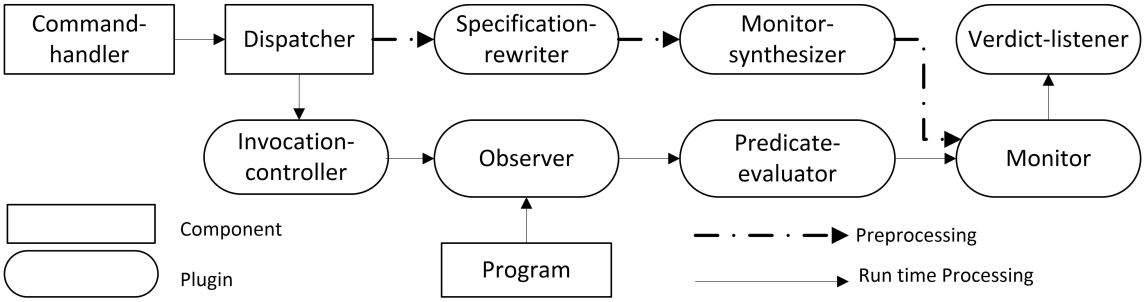


Figure 3.1: RitHM Architecture and Basic Control flow

and development of runtime verification techniques. The contributions of this part of thesis are:

- The design and architecture of RitHM, a comprehensive framework for runtime verification
- Various novel features of RitHM's design and the reasoning for the associated design decisions
- Evaluations of architectures using architecture trade-off analysis method (ATAM)
- An empirical evaluation of RitHM by monitoring of real-world datasets

3.2 RitHM Framework

In this section, we describe the details about architecture and design of RitHM. Moreover, we describe important design features such as the interoperability between RitHM's monitors. Additionally, RitHM can operate in *outline* [52] mode, where an execution trace is sent over a network, and RitHM listens for the events using TCP sockets. This mode permits a remote instance of RitHM to be configured by a client, where the client sends monitoring commands to the RitHM instance running as a server. We also describe the details of RitHM's built-in plugins.

3.2.1 RitHM Architecture

Figure 3.1 depicts the important components of RitHM along with details of typical control flow among them. We further describe the details of various components.

Command-handler: This component receives input parameters, and it validates the input parameters for various constraints. Some of the input parameters are described in the example presented later in this chapter. The Command-handler then passes the relevant details to other components such as the Plugin-loader and the Dispatcher for further processing.

Plugin-loader and Dispatcher: Plugin-loader attempts to load various types of RitHM plugins based on the supplied input. Various types of plugins and their functions are described in later parts of this section. Dispatcher instantiates the loaded plugins with respective input parameters. It then passes the control to Invocation-controller, which co-ordinates the invocation of the specified monitor.

Invocation-controller: This component's responsibility includes invoking the synthesized monitors in conjunction with other required plugins. While events are extracted from a program by Observer, Invocation-controller invokes Monitor w.r.t. certain conditions and passes a reference of Observer's *buffer* to Monitor. The invocation itself can be performed using various schemes of overhead-control such as event-triggered RV or buffer-triggered RV.

This component is an extensible component to allow a user to reuse and extend various overhead-control schemes. The default implementation of this components can perform buffer-triggered monitoring (BTM) [57] and event-triggered monitoring, where Monitor is invoked for each event extracted in the buffer.

In case of *online* monitoring, the programs being monitored utilizes Invocation-controller *directly* using RitHM's APIs to control the invocation of Monitor.

Observer: This component handles the functionality of transforming the extracted events into RitHM's internal format. In case of *offline* monitoring, the user specifies a path to a file containing the events. For *online* monitoring, an instrumentation tool extracts the events from a running program and the events can be passed to RitHM by using the APIs of respective Observer plugin. The built-in implementations of this plugin transform data from XML and CSV formats. This component is designed to be a RitHM plugin so that a user can write implementations of this plugin in order to monitor events in various external formats.

Specification-rewriter: This component parses the supplied specification and validates their syntax. Next, it constructs a parse-tree for each of the specifications, and the parse-trees are utilized by Monitor-synthesizer for constructing Monitor. Further, this plugin also handles the responsibility of translating the supplied specifications in case needed.

Here, we note one of the important design features of RitHM, where the Specification-rewriter is decoupled from Monitor-synthesizer. This decoupling helps RitHM to utilize an implementation of Monitor-synthesizer with different implementations of Specification-rewriter, and vice versa. This allows a logician to develop a high-level specification language, which can then be internally *compiled* into a low-level specification language for which a monitoring algorithm exists. For example, RitHM contains a built-in implementation of Specification-rewriter for *verbose* Linear Temporal Logic (VLTL), which has a set of verbose tokens in its syntax. These tokens of VLTL can be directly mapped to that of Ltl. The rewriter service internally translates VLTL formulas into Ltl formulas before they are passed on to the Monitor-synthesizer plugin for Ltl.

Monitor-synthesizer: This component constructs a Monitor using a set of monitoring procedures for the supplied specifications, which are validated by Specification-rewriter. At run time, Monitor receives events from Observer. Invocation-controller *controls* the interaction between Observer and Monitor. A user can develop different implementations of this component to monitor specifications in different types of logical formalisms. Monitor utilizes other components such as Verdict-listener and Predicate-evaluator.

Verdict-listener: This component, which is also a RitHM plugin, gets notified by Monitor, when there is a change in the truth-values of specifications. A user can create custom implementations for different purposes. For example, when Monitor detects a violation of the correctness criteria, an implementation of this plugin can steer the program back into a correct state.

Predicate-evaluator: This component consumes events and finds the truth-values of the predicates in first-order logic (resp. propositions in propositional logics). Since, the evaluation of predicates (resp. propositions) differs for different use-cases, a user can provide a custom implementation in various ways. The built-in implementations of this predicate allow a user to perform an evaluation using JavaScript and Lua scripts. Such scripts consume an event, i.e., a program-state in RitHM's standard format, and an evaluation of scripts assigns values to different predicates used in the respective specifications.

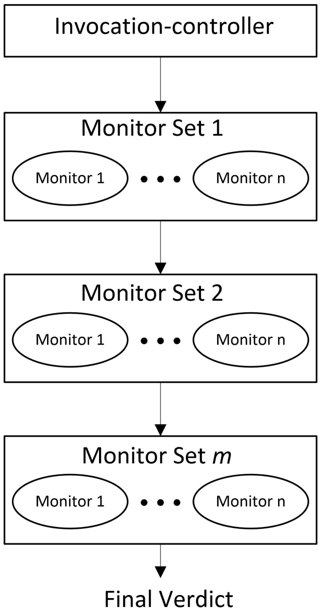


Figure 3.2: RitHM Interoperability of Monitors

The scripts need to be designed such that the valuation of predicates is returned in RitHM's standard format.

As described earlier, many components of RitHM are supported as plugins. Consequently, RitHM could be customized for different use-cases.

3.2.2 Interoperability of Monitors in RitHM

RitHM is designed such that different monitors can be used in conjunction with each other. This enables a reuse of efficient algorithms for monitoring various fragments of a logical formalism. Further, this functionality allows decomposition of a complex specification into various simple specifications, which can be monitored by different monitoring algorithms. Consequently, this feature facilitates a development of monitors for the extensions of existing logical formalisms.

This design feature of RitHM works alongside the rewriter interface and monitoring algorithms. This feature enables a user to develop high-level specification languages, and the interoperability feature enables him to combine existing implementations of Monitor interface.

As shown in Figure 3.2, Invocation-controller starts first set of monitors, and a set of specifications are processed. The verdicts of the monitors are saved. The next set of monitors utilizes these saved verdicts as inputs. This ‘piping’ process continues until the last set of monitors produces final verdicts.

This feature of RitHM is similar to that of LOLA [24]. LOLA saves the verdicts of the sub-formulas into variables. These saved verdicts are utilized for determining the verdict of a top-level formula. However, it is not specified whether LOLA can save the verdicts of monitors for specifications, which are expressed in a specification language different from LOLA’s own language. On the contrary, RitHM allows interoperability between monitors of different logical formalisms, and this enables a user to develop extensions of different types of logical formalisms. For example, one could ‘wrap’ an existing implementation of a monitor for MFOTL with additional aggregation operators by developing a Monitor-synthesizer plugin to handle the additional aggregation operators.

When the interoperability feature is used, RitHM internally constructs a set of tree structures. In each of these tree structures, the leaves denote the monitors executed in the first stage. The monitors in the higher levels of the tree utilize the verdicts produced by the monitors in the previous levels. Hence, if a monitor M_{i+1} in level $i+1$ utilizes the verdicts produced by a monitor M_i in level i then it is required that M_i is forward interoperable with M_{i+1}). This is denoted by $(M_i \rightharpoonup M_{i+1})$.

Definition 1 (Interoperable Monitors)**.** *Let $\mathbb{B}_1$ be the output truth-domain, i.e., the set of possible truth-values produced by M_1, a Monitor Plugin. Further, $\mathbb{B}_2$ denotes the input truth-domain, i.e., the set of possible truth-values processed as an input by M_2, which is another Monitor Plugin.*

M_1 is forward interoperable *with M_2, iff. $\mathbb{B}_1 \subseteq \mathbb{B}_2$, i.e., the set of possible truth-values produced by M_1, is valid as an input for M_2.*

3.2.3 RitHM for RV in Outline Mode

RitHM can be run as a server as shown in Figure 4.1. Different clients can utilize the RitHM client library to connect to a RitHM server. This library implements a simple communication protocol. The clients can be programs, which run on remote machines, or they could be other utilities, which connect to a RitHM server for offline monitoring.

Clients can connect to a RitHM instance in a secure manner using SSL (Secure Sockets Layer) or by using plain sockets for un-secure connections. The clients can configure the RitHM instance using a set of configuration commands. The client then sends the execution trace to the RitHM instance, which buffers the trace and monitors process it.

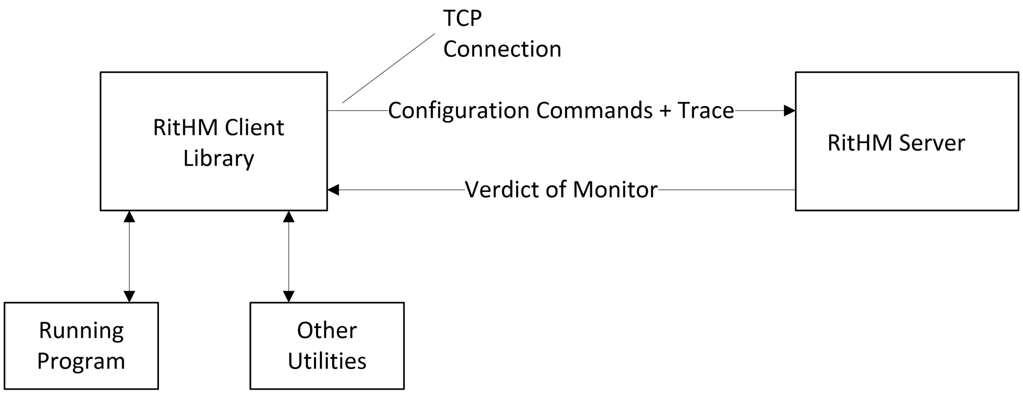


Figure 3.3: RitHM Server

3.3 RitHM Plugins

In this section, we describe built-in plugins for various components of RitHM. These plugins implement various algorithms developed in the area of runtime verification for different logical formalisms.

3.3.1 Observer Plugins

RitHM provides two built-in plugins for data-extraction and the transformation of event data into RitHM's standard format. These plugins extract data from XML and CSV formats respectively. Once extracted, the event data is transformed into RitHM's standard format.

The XML syntax expected by the built-in plugin for XML is as shown in Listing 3.1. Each `State` tag in the XML data corresponds to a distinct event. A `Key` tag is used to define the names of various parameters for an event. A `Value` tag provides the value of a event parameter defined by the previous `Key` tag. The plugin also expects a `timestamp` tag, which encloses the value of the timestamp for the corresponding event. The value is used for monitoring of specifications described using timed-temporal logics such as MTL.

The CSV syntax expected by the CSV plugin is as shown in Listing 3.2. Every line of a valid CSV file contains a set of key-value pairs, which are separated by comma. The plugin also expects that `timestamp` as one of the keys in every line. A new line indicates a start of details for the next event.

```
<Trace>
```

```
<State>
<Key>keyname_1</Key>
<Value>keyvalue_1</Value>
<Key>keyname_2</Key>
<Value>keyvalue_2</Value>
...
<Key>keyname_n</Key>
<Value>keyvalue_n</Value>
<timestamp>01</timestamp>
</State>
<State>
...
</State>
...
</Trace>
```

Listing 3.1: Expected XML Syntax

```
keyname_11=value_11,...,keyname_1m=value_1m
keyname_21=value_21,...,keyname_2m=value_2m
...
...
keyname_n1=value_n1,...,keyname_nm=value_nm
```

Listing 3.2: Expected CSV Syntax

3.3.2 Specification-rewriter Plugins

RitHM provides built-in plugins for parsing and rewriting specifications in the following logical formalisms: Ltl [14], PTLTL [40], MTL [76], A fragment of FOLTL [56], VLTL, etc. We further illustrate the syntax of logical formalisms for a few selected plugins.

Ltl [14]: The Specification-rewriter plugin for Ltl uses below syntax for validating Ltl specifications. Below is an example of an Ltl specification.

- `ThreadCreated`− >`<>ThreadReady`: This specification expresses an idea that when a *thread* is created by a process, it is *eventually* in ‘Ready’ state.

```
LTL  :=
    "TRUE"
  | Event
  | "!" LTL             // Not
  | "[]" LTL            // Globally
  | "<>" LTL            // Eventually
  | "X" LTL             // Next
  | LTL "&&" LTL        // And
  | LTL "||" LTL        // Or
  | LTL "->" LTL        // Implies
  | LTL "U" LTL         // Until
  | "FALSE"
```

Listing 3.3: LTL Syntax

PTLTL [40]**:** The Specification-rewriter plugin for PTLTL uses below syntax for validating PTLTL specifications.

```
PTLTL :=
    "TRUE"
  | Event
  | "!" PTLTL           // Not
  | "[*]" PTLTL         // Globally in past
  | "<*>" PTLTL         // Eventually in past
  | "X*" PTLTL          // Previously
  | PTLTL "&&" PTLTL    // And
  | PTLTL "||" PTLTL    // Or
  | PTLTL "->" PTLTL    // Implies
  | PTLTL "S" PTLTL     // Since
  | "FALSE"
```

Listing 3.4: PTLTL Syntax

Below is an example of an PTLTL specification.

- $[*](\texttt{AlaramRaised} - >< * > \texttt{Fault})$: This specification expresses an idea that it is always the case that when an *alarm* is raised, a *fault* had occurred in the past.

MTL: The parser for MTL uses below syntax for validating MTL specifications.

```
MTL  :=
    "TRUE"
  | Event
  | "!" MTL             // Not
  | "[]{x,y}" MTL       // Bounded Globally
                        // with time interval
                      // [x,y]
  | "<>{x,y}" MTL     // Bounded Eventually
                      // with time interval
                      // [x,y]
  | MTL "&&" MTL      // And
  | MTL "||" MTL      // Or
  | MTL "->" MTL      // Implies
  | MTL "U{x,y}" MTL  // Bounded Until with
                      // time interval
                      // [x,y]
  | "FALSE"
```

Listing 3.5: MTL Syntax

Below is an example of an MTL specification.

- [](CANMsgSent− ><> {0, 2}CANMsgDelivered): This specification expresses the idea that it is *always* the case that when a *message* is *sent* via a CAN (Controller Area Network) bus, it is *eventually* delivered within 2 Milliseconds.

3.3.3 Monitor-synthesizer Plugins

A set of built-in implementations of the Monitor Plugin are available for the following logical formalisms: Ltl, PTLTL, MTL, a fragment of FOLTL and Regular Expressions. The plugin for monitoring Ltl utilizes the algorithms developed by Bauer at al. [13, 14]. Bauer et al. have developed implementation of these algorithms in the form of `ltl3tools` utility. The plugin for Monitoring specifications expressed using regular expressions uses an automaton library, which is developed by Møller et al [59]. The plugin for monitoring MTL use variants of the algorithms described in [10, 35]. The Monitor plugin for FOLTL implements a variant of monitoring algorithm [56] developed by Medhat et al.

3.3.4 Other Plugins

The default Predicate-evaluator plugin assumes that an event trace provided by the Observer plugin contains the evaluation of the associated predicates or propositions. Further, a set of implementations for evaluating predicates (resp. propositions) using scripts written in ‘Lua’ and ‘JavaScript’ are also available. The default Verdict-listener plugin outputs the changes in verdicts of monitors at the standard output. The default implementation of Invocation-controller component allows configuring a Monitor plugin for *buffer-triggered* and *event-triggered* runtime verification.

Proposition	**Evaluation Condition**
p	$currentRPM > 4,000 \wedge \mathrm{prevRPM} \leq 4,000$
q	$currentRPM \leq 4,000 \wedge \mathrm{prevRPM} > 4,000$
r	$\lambda <= 1.2 \wedge \lambda >= 0.8$
s	$\lambda > 1.2$

Table 3.1: Propositions Used in the Example

3.3.5 RitHM Usage

RitHM is designed for users, who develop a set of specifications for verification. Further, we assume that RitHM's users have basic programming knowledge. RitHM's configuration can be described using command-line interface or through configuration files. Using RitHM through configuration files allows a user to reuse several features of RitHM and only configure the required parts of a RitHM instance.

We illustrate RitHM's usage using an example. In this example, we monitor a trace of an Engine Control Unit (ECU) used in [57]. The trace, which is a CSV file, contains various parameters. The properties are defined using the parameters namely 'lambda', 'currentRPM' and 'prevRPM'. The value of the 'lambda' parameter indicates the output of the lambda sensor of the ECU. This value is equal to the air-to-fuel ratio of the engine. The parameters 'currentRPM' and 'prevRPM' indicate the value of engine's revolutions per minute in current reading and previous reading respectively.

The configuration file used in the example is shown in Listing 3.6. We intend to verify two LTL specifications on the execution trace. Additionally, we use LTL_3 semantics. The evaluation of propositions is done using 'Lua' script, which is shown in Listing 3.7. This script, which uses the value of the parameters in the execution trace as an input, evaluates the propositions used in the two LTL formulas. The script sets the value of a proposition to '1', when the associated condition is true. As shown, the script evaluates the values of four propositions w.r.t. the conditions, which are shown in Table 3.1.

The two LTL properties are chosen from Dwyer's patterns of LTL formulas, and these safety properties specify the conditions on the value of the 'lambda' parameter w.r.t. the variation in the value of engine's revolutions per minute. An instance of RitHM is started using `RitHMBrewer` utility as shown in Listing 3.8. As the configuration file does not specify a plugin of type 'Verdict-listener', RitHM uses default plugin, which outputs the changes in the verdict of a specification at the console. The two safety formulas are not violated by the trace. Thus, the verdicts are shown as 'unknown' as per LTL_3 semantics.

```
dataFile=rithm_csv_engine_data.csv
specParserClass=LTL
specifications="[]((q&&! r&&<>r)->(p U r))
                []((q&&! r&&<>r)->(! s U r))"
monitorClass=LTL3
traceParserClass=CSV
predicateEvaluatorType=lua
predicateEvaluatorScriptFile=exampleEvaluator.lua
```

Listing 3.6: Example of a RitHM Configuration File

```
local function main()
  retval={}
  if tonumber(currentRPM) > 4000 and
     tonumber(prevRPM) <= 4000 then
    retval["q"] = tostring(1)
  end
  if tonumber(currentRPM) <= 4000 and
     tonumber(prevRPM) > 4000 then
    retval["r"] = tostring(1)
  end
  if tonumber(lambda) <= 1.2 and
     tonumber(lambda) >= 0.8 then
    retval["p"] = tostring(1)
  end
  if tonumber(lambda) > 1.2 then
    retval["s"] = tostring(1)
  end
  return retval
end
return main()
```

Listing 3.7: Evaluating Propositions using Lua Scripting

```
user1@comp1:~$ RitHMBrewer -configFile rithm.properties
[]((q\&\&! r\&\&<>r)->(p U r)):=Unknown
[]((q\&\&! r\&\&<>r)->(! s U r)):=Unknown
user1@comp1:~$
```

Listing 3.8: Using RitHMBrewer

Chapter 4

Considerations for RitHM's Design

In this chapter, we enumerate different considerations, which have driven our design decisions and the associated features. These decisions typically focus on achieving efficiency in RV techniques in terms of resource utilization and latency. Further, the design decisions also address the problem of development of expressive logical formalisms and their high-level abstractions. Additionally, we provide an evaluation of RitHM's architecture using architecture trade-off analysis method (ATAM) [48]. We also evaluate RiTHM[1], MOP and Larva using ATAM. The evaluations highlight the importance of unique design features of RitHM.

4.1 The Front-end and Back-end of RitHM

We illustrate one of the most important design features of RitHM. Figure 4.1 shows high level view of RitHM's architecture, where the front-end components are separated from back-end. As described earlier in Section 1, the front-end handles the tasks of processing specifications and configurations of a monitor. Typically, the preprocessing happens at the front-end, but the back-end runs alongside a program, which is under verification. So, the back-end needs to run optimized algorithms for reducing the overhead caused by RV. This involves controlling the invocation of a monitor by different schemes such as BTRV, TTRV [17], control-theoretic schemes [57], etc. Further, different efficient monitoring algorithms, which use parallel [5, 15, 35, 56] verification techniques, can be run at the back-end. On the other hand, expressivity is one of the important aspects for the

[1] RitHM's predecessor

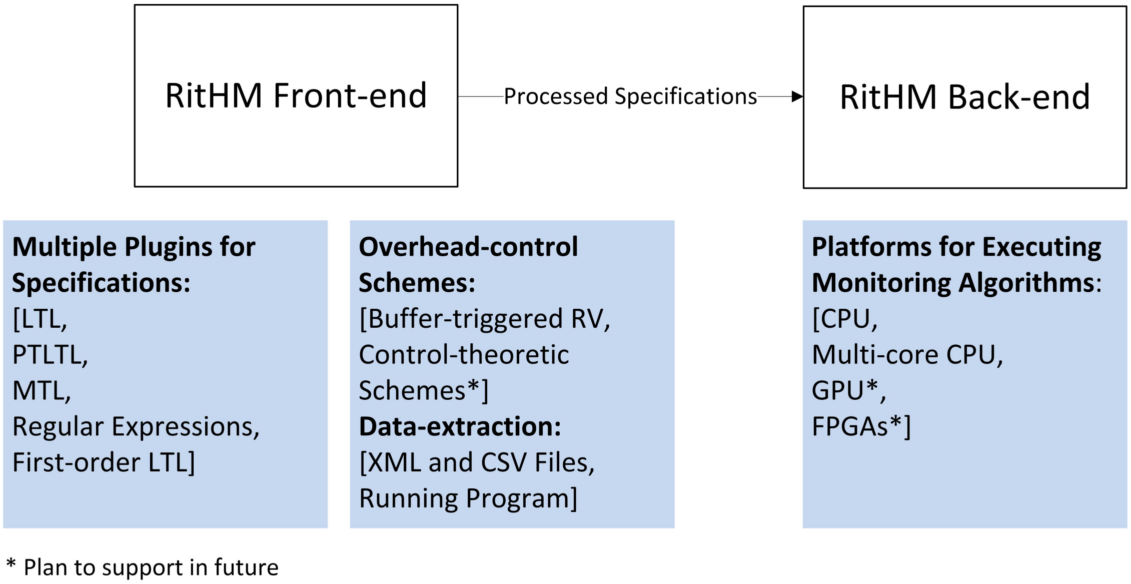


Figure 4.1: Summary of functions

front-end, which processes specifications. The front-end executes *offline* algorithms, which validate, translate or transform a set of specifications. Considering these factors, we developed RitHM's architecture, where the components such as Invocation-controller, Observer, Monitor, Predicate-evaluator and Verdict Listener constitute the back-end. Further, the components such as Command-handler, Dispatcher and Specification-rewriter constitute the front-end of RitHM.

As most of the components of the front-end and back-end are supported as plugins, a user can utilize different combinations of the front-end plugins and back-end plugins. Further, as introduced previously, to be adapted widely, the RV techniques need to improve in their front-end aspects, where user-friendly ways of defining specifications are needed [2]. A segregation of the front-end components from the back-end components facilitates integration of complex *offline* algorithms, which perform specification mining, translation and transformation of specifications. These algorithms are separated from *online* monitoring algorithms, which are required to be efficient in terms of performance and resource utilization.

4.2 Support for Multiple Logical Formalisms

Figure 3.1 shows that the Specification-rewriter component of RitHM is supported as one of the plugins. This feature is important because of the following reasons.

Firstly, there are various types of logical formalisms, and they support different use-cases and scenarios. For example, LTL serves to specify properties where an explicit notion of time is not needed, and the knowledge about the ordering of *events* in a system is enough for verification. On the other hand, MTL can be used to specify specifications of time-sensitive systems, where a time interval between different *events* of a system needs to be explicitly bounded. Hence, such logics are relevant for describing specifications of real-time systems. Further, logical formalisms such first-order LTL and MFOTL, which support parametric monitoring, are useful for verification of systems with multiple objects. Thus, a typical RV setup needs to support mutliple logical formalisms.

Secondly, the evolution of logical formalisms shows a trend, where more complex and expressive logical formalisms are created by extending previously developed formalisms. Some examples of such extensions are: addition of aggregation operators to MFOTL for MONPOLY, development of 'Counting' quantifier for first-order LTL by Bauer et al. [11], 'Counting' semantics for first-order LTL by Medhat et al. [56], etc.

Thirdly, RitHM's rewriter interface supports various APIs, which allow translating or transforming specifications. This allows developing higher level abstractions of existing logical formalisms. Dwyer et al. [29] have identified commonly used patterns of LTL specifications. In this work, they developed mappings of high-level descriptions of the patterns to the corresponding LTL formulas. For example, 'P is *false* **before** R becomes *true*' is mapped to $\Diamond R \rightarrow \neg P \, U \, R$. Here, P and R are atomic propositions. Such patterns increase the *usability* of the logical formalisms. RitHM provides APIs to support such translation of specifications.

Finally, there are different specifications, which could be written using multiple logical formalisms. LTL and PTLTL are found to be equally expressive [33], but PTLTL is proven to be more succinct than LTL in terms of the size of a temporal formula. There are different monitoring algorithms available for LTL and PTLTL. However, the monitoring algorithm for LTL [14], which uses a finite state Moore machine, is more efficient than that of the rewriting based algorithm [41] because later requires extra memory buffers to preserve the evaluations of all subformulas while arriving at a verdict of the corresponding top-level PTLTL formula. Hence, the time complexity of the monitoring algorithms is $O(m \times n)$, where m is the size [40] of the PTLTL formula and n is the number of events.

We performed an experiment with a set of safety formulas of different sizes. For the

No.	**PTLTL Formula**	**LTL Formula**
1	$\boxdot\,(a_2 \rightarrow (\neg a_1 \wedge \odot a_1)$	$\Box(\bigcirc a_2 \rightarrow (\neg \bigcirc a_1 \wedge a_1))$
2	$\boxdot\,(a_2 \rightarrow (a_1 \wedge \neg \odot\ a_1))$	$\Box(\bigcirc a_2 \rightarrow (\bigcirc a_1 \wedge \neg a_1))$
3	$\boxdot\,(a_1)$	$\Box(a_1)$
4	$\bigwedge\limits_{i=1}^{5}(\boxdot\,(a_{i+1} \rightarrow$ $(a_i \wedge \neg \odot\ a_i))$	$\bigwedge\limits_{i=1}^{5}(\Box(\bigcirc a_{i+1} \rightarrow$ $(\bigcirc a_i \wedge \neg a_i)))$

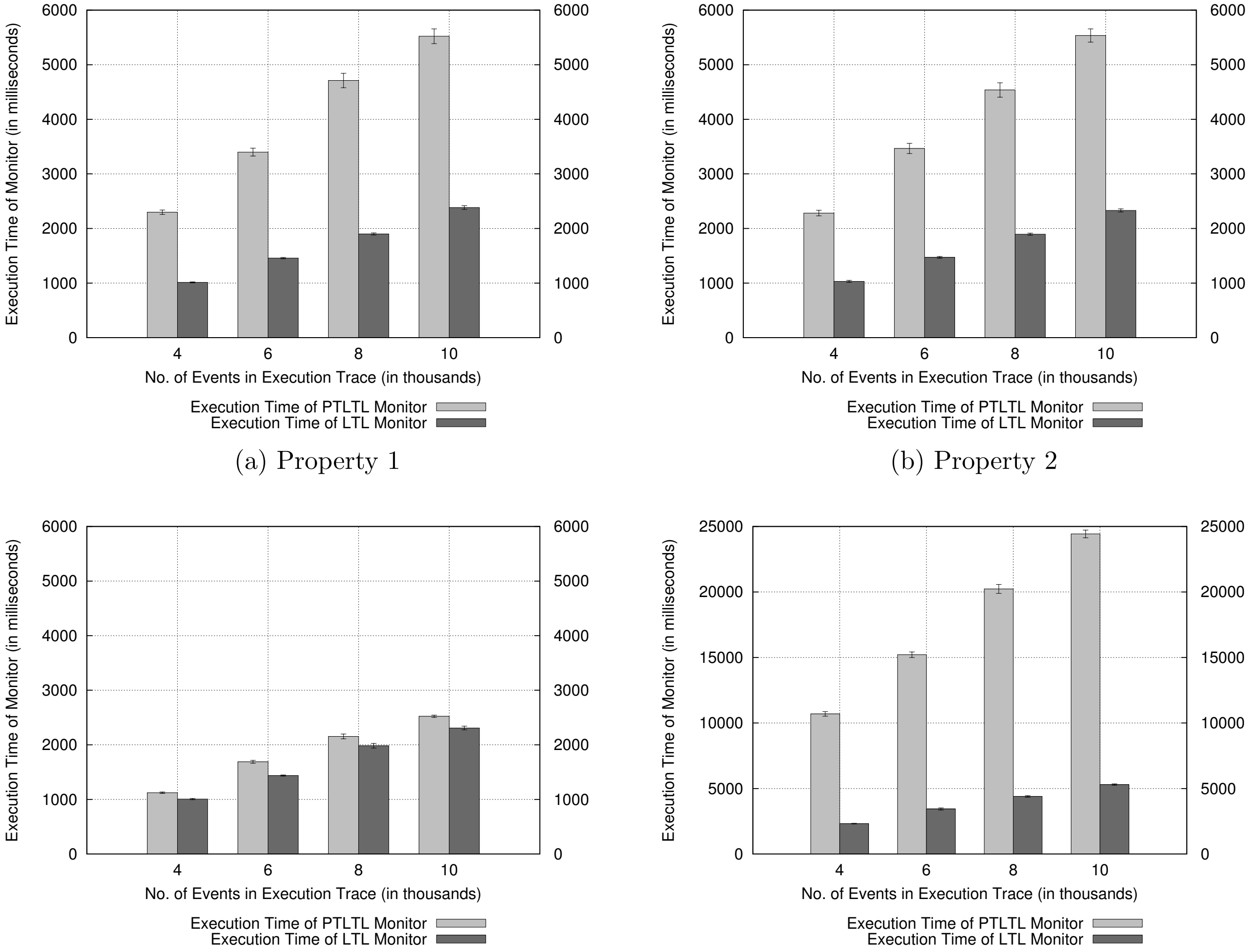
Table 4.1: PTLTL and Ltl formulas

experiments, we used a single eight core machine equipped with the Intel i7-3820 CPU at 3.60GHz and 31.4Gb of RAM. The machine runs Ubuntu 14.04 LTS 64 bit. Further, each of the safety formula was described using two logical formalisms namely PTLTL and Ltl. The formulas are shown in Table 4.1. The PTLTL operators 'previously' and 'globally in the past' are denoted by $\odot$ and $\boxdot$ respectively. We also note that over a finite trace, an appropriate initialization of truth-values of propositions is required while evaluating sub-formulas with 'previously' operator of PTLTL [40].

We compared the overhead of PTLTL and LTL monitors in terms of their execution time. Figure 4.2 shows the comparison of overhead of PTLTL and Ltl w.r.t. the variation in the number of events. The x-axis shows the number of events in traces, and the y-axis shows the execution time of monitors in milliseconds. As the size of the formula increases, the Ltl monitor outperforms the PTLTL monitors by an increasing margin. For Property 3, the difference between the execution times is not significant. However, it is significant for Property 4, where the size of the formula is bigger than that of Property 3. The findings of this experiment suggest that a support for multiple logical formalisms in front-end provides flexibility to a user to choose appropriate configurations for respective use-cases. Additionally, a user can evaluate multiple monitoring algorithms by encoding a specification into multiple logical formalisms.

4.3 Support for Monitor Plugins and Multiple Backends

The Monitor-synthesizer component of RitHM is supported as a plugin. RitHM's Monitor-synthesizer plugins generate monitors, which implement monitoring algorithms for a logical formalism or its particular fragment. Hence, there can be mutliple monitors available



(a) Property 1

(b) Property 2

(c) Property 3

(d) Property 4

Figure 4.2: Comparison of Monitoring Overhead for PTLTL and LTL Formulas

for a logical formalism, and a user can choose the optimal monitor as required by the respective use-case. Further, such monitors can be run on a back-end such as multi-core CPU depending upon whether a parallel algorithm exists for performing efficient verification.

A use of mutli-core CPU, GPU and FPGAs has been recently shown to be useful to runtime monitoring [15, 56, 64]. Such use causes less-intrusive verification of a system as the associated monitor introduces less jitter in the system's performance because of lesser proportion of shared resources compared to that of a monitor, which runs alongside the system on same processor. Further, such monitor also provides isolation, which ensures that the monitor is less affected by failures of underlying hardware on which the system runs. A use of mutli-core CPU can reduce latency in monitoring as a result of parallel-processing. Moreover, it facilitates efficiency in terms of power consumption [34].

For parametric monitoring, it is often the case that a large number of monitors are created for individual objects found in an execution trace. Additionally, in many cases, such monitors can be created in a tree structure [56], where the verdict of monitors in level i depends upon the verdict of monitors at level $i + 1$ in the tree.

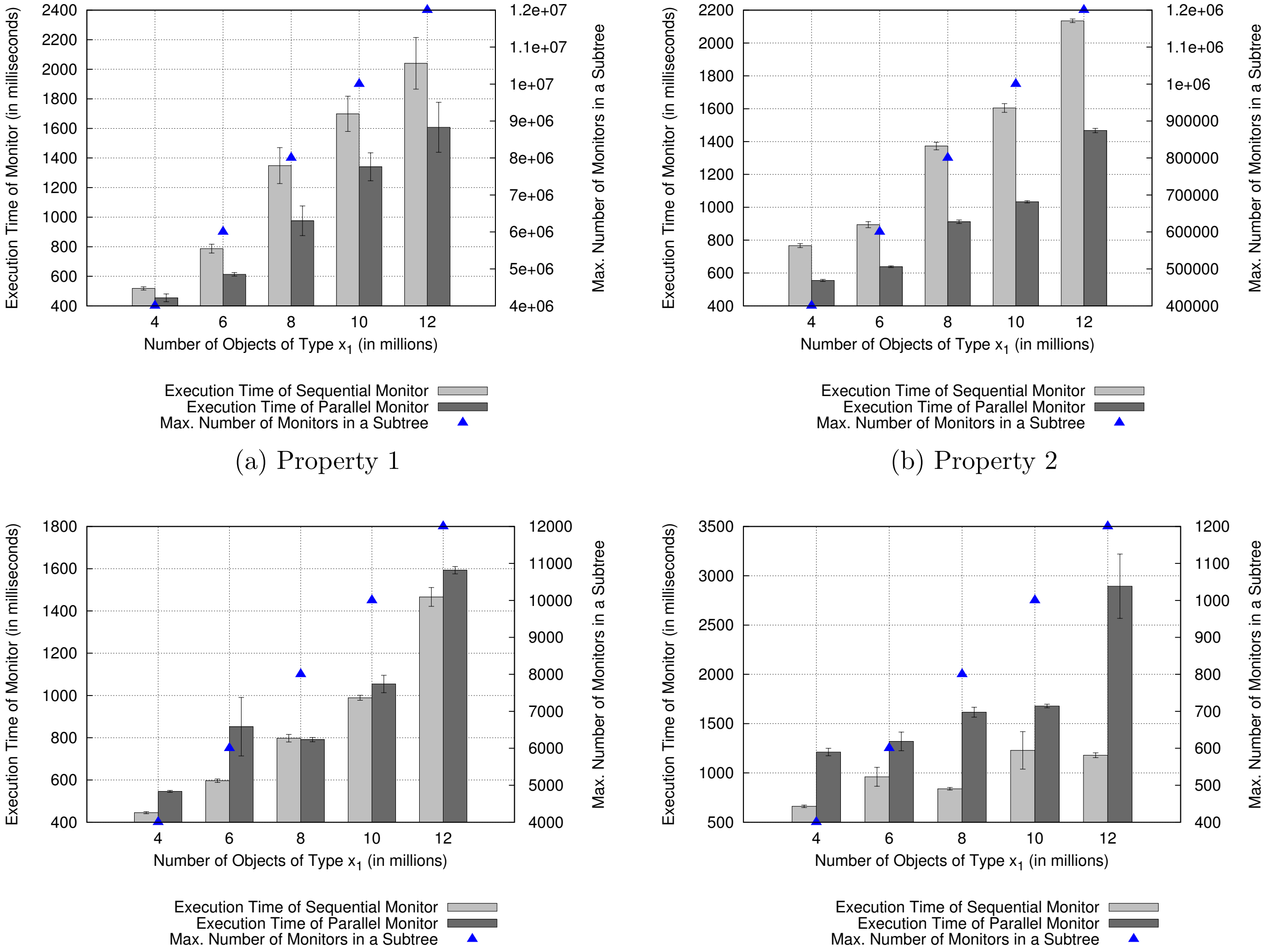
For example, a formula $\forall \mathit{pid}\, \forall \mathit{fd} \cdot \mathit{usedBy}(\mathit{pid}, \mathit{fd}) \rightarrow (\mathit{open}(\mathit{pid}, \mathit{fd}) \rightarrow \Diamond \mathit{close}(\mathit{pid}, \mathit{fid}))$ needs to be evaluated using two types of objects namely *pid* and *fd*. This formula expresses the property that for all processes and all file descriptors, if a file descriptor is used by a process, then an *open* action on the file descriptor is eventually followed by a *close* action. A monitor for this first-order Ltl formula is created using a tree-structure of monitors for the underlying Ltl formula. The structure consists of Ltl monitors for every *fd* used by each process. All such monitors, which belong to a particular *process*, are the children of the parent monitor for the respective *pid*. The parent monitor maintains the verdict of the Ltl formula for its sub-tree. The top level monitor, which holds the verdict of evaluation for the first-order formula, is the parent of all monitors for individual processes. The verdict of the top-level formula is determined by performing an aggregation operation on the verdicts of all child monitors. This operation, whose definition depends upon the type of quantifier, can be performed in parallel, and such parallelization can be efficient provided a large number of monitors exist for a monitor in its subtree. Parallel algorithms for aggregation operations are proved to be performance efficient for processing large scale data [60].

To verify the feasibility of this approach, we implemented a variant of the monitoring algorithm [56] described by Medhat et al. for a fragment of first-order Ltl. The environment described in Section 4.2 is also used for this experiment. We verified the performance of sequential and parallel monitoring algorithms in terms of the execution time for performing the aggregation operations to produce the final verdict for a first-order formulas. We note that the Ltl monitoring functionality is identical for the parallel and sequential monitors. However, the implementation of aggregation operation for the 'existential' and 'universal' quantifiers differs in terms of the implementation and usage of multi-cores. We experimented by generating the monitors for following properties using RitHM's monitor generator for the aforementioned fragment of first-order Ltl.
Property 1: $\forall x_1 \cdot a(\mathit{x_1}) \rightarrow \Diamond b(\mathit{x_1})$.
Property 2: $\forall y_1\, \forall x_1 \cdot \Box(a(\mathit{x_1}, \mathit{y_1}) \rightarrow (b(\mathit{x_1}, \mathit{y_1})\, \mathbf{U}\, c(\mathit{x_1}, \mathit{y_1})))$

Property 3: $\forall y_1 \forall x_1 \cdot \Diamond(a(x_1, y_1) \wedge (\Diamond b(x_1, y_1)))$
Property 4: $\forall z_1 \forall y_1 \forall x_1 \cdot \Diamond(a(x_1, y_1, z_1) \rightarrow (\neg b(x_1, y_1, z_1)\, \mathbf{U}\, a(x_1, y_1, z_1)))$
We compared the performance of parallel and sequential monitors for different size of traces by varying the number of objects for each object type referred in the quantifiers. We use 20 replicates for the experiments. For Property 1, the number of child monitors of the top-level monitors is same as that of the number of objects of type 'x_1'. For Property 2, the number of objects of type 'y_1' is set to 10. For Property 3, the number of objects of type 'y_1' is set to 1000. For Property 4, the number of objects of type 'y_1' is set to 100, and the number of objects of type 'z_1' is set to 100. The number of objects of type 'x_1' is varied from 4,000,000 to 12,000,000. We intend to verify the required size of data for which the benefits of parallel algorithm can be achieved. We note that the parallel implementation performs additional work in terms of the number of executed instructions. Further, the overhead of fork-join model of Java threads may cause the parallel implementation to actually perform slower than the sequential algorithm provided the volume of data is less than a certain threshold.

As shown in Figure 4.3, the results are shown for the comparison of monitoring overhead for the four properties. The x-axis shows the number of objects of type 'x_1'. The y-axis on left shows the execution time of the monitor to produce a verdict by performing aggregation operations, i.e., the time to perform aggregation operation(s). We note that the time for evaluating the underlying Ltl formula is identical for both the variants. The y-axis on right provides the maximum number of monitors used for the evaluation of a quantifier. The results show that for Property 1 and Property 2, the parallel implementation outperforms the sequential implementation. For these two properties, the number of monitors used for the evaluation of quantifiers is high. Thus, the parallel reduction outperforms the sequential reduction. The speedup achieved varies primarily due to the effects of garbage-collection operation and the extra overhead of spawning threads.

On the other hand, for Property 3 and Property 4, the maximum number of monitors used for the evaluation of quantifiers is not big enough to achieve the benefits of parallelism, and the extra-overhead of executing multiple threads affects the performance. Thus, the sequential implementation outperforms the parallel implementation. These experiments highlight that the parallel algorithm can be useful for monitoring traces with a large number of objects. Consequently, such algorithms can be potentially applied for monitoring large-scale cloud applications. In our experimental configuration, we found that when an aggregation operation, executed during an evaluation of a first-order formula, is performed on a large number of objects, the parallel monitoring algorithm consistently outperforms the sequential variant.



(a) Property 1

(b) Property 2

(c) Property 3

(d) Property 4

Figure 4.3: Comparison of Monitoring Overhead for Parallel and Sequential Algorithms

4.4 Interoperability between RitHM's Monitors

RitHM facilitates interoperability between different monitoring algorithms. This feature enables a functionality where a Monitor plugin's output can be utilized as an input variable by another Monitor plugin. An output of a Monitor plugin is a truth-value from a *truth-domain*, which is a partially ordered set with an upper bound and a lower bound. Hence, output of Monitor plugins may necessarily not be boolean values. For example, a

LTL_3 monitor [14] produces an output in truth-domain $\mathbb{B}_3 = \{\top, \bot, ?\}$. Hence, the interoperability feature requires that the Monitor plugins are compatible w.r.t. the processing of truth-domains. The design decision to facilitate the interoperability between monitoring algorithms is driven by following factors.

Firstly, a specification can be divided into sub-formulas, and each of these sub-formulas could be monitored using different efficient algorithms. In this case, the respective sub-formulas could be processed by different Monitor plugins of RitHM. The can improve performance of monitoring as efficient algorithms for respective fragments of logical formalisms can be reused. For example, a formula expressed using MTL may contain a sub-formula, which could be monitored using the recursive definition based algorithm [35]. This algorithm is memory efficient as compared to that of the standard algorithm for monitoring MTL because it is based on recursive definitions and requires *bounded* buffer. The interoperability feature ensures that the verdicts of different monitors for different sub-formulas can be seamlessly combined to derive the final verdict.

Secondly, the interoperability feature enables extension of monitoring algorithms. As described previously, different logical formalisms have evolved with extension or augmentation of their features. Hence, corresponding monitoring algorithms can also be extended by reusing features of other algorithms by utilizing the interoperability feature.

While interoperability can help efficient monitoring, the performance benefits vary for different use-cases. For example, if a final verdict depends on verdicts of different sub-formulas, then a reduction in latency by one of the monitoring algorithms does not contribute for corresponding reduction in latency to arrive at the final verdict. However, the interoperability feature can be useful for achieving efficiency in memory utilization in such cases.

4.5 Evaluation of architectures using ATAM

Various methods such as architecture trade-off analysis method (ATAM) [48], software architecture analysis method (SAAM) [47] are used for structured of evaluation of software architectures. ATAM has been a leading method for evaluating software architectures.

ATAM evaluation works w.r.t. the business drivers of an architecture. The business drivers are in turn used to identify the non-functional requirements, which an architecture needs to exhibit. The non-functional requirements are referred to as *quality attributes*. ATAM helps to predict the consequences of architectural decisions for attaining the specified *quality attributes* at an early stage of a project. Thus, ATAM provides trends w.r.t.

the existence of *risks*, *non-risks*, *sensitivity points* and *trade-off points* in an architecture. The description of these notions is as follows.

- Risks are the design decisions, which might create problems for some quality attributes.
- Non-risks are good design decisions, which enables an architecture to meet certain quality attribute(s).
- Sensitivity points are the design alternatives for which a slight change in the alternative can cause a significant impact on a quality attribute.
- Trade-off points indicate an existence of design alternatives, which affect multiple quality attributes.

In this sub-section, we provide evaluations of RitHM's architecture and architectures[2] of RiTHM [63], the predecessor of RitHM, MOP [58] and Larva [22]. These evaluations highlight the benefits of various unique design features of RitHM. We note that comprehensive details about the architectures are not available in some cases. However, the purpose of the evaluation is to provide trends w.r.t major architectural decisions. To our knowledge, our evaluation is the first architectural analysis of frameworks and tools in the RV community.

Although the business drivers of different RV tools and frameworks are different, the requirements w.r.t. the quality attributes are often identical. CSRV'14 evaluated various RV tools and frameworks based upon their expressivity and efficiency. This evaluation criteria of CSRV'14 suggests that modifiability is one of the important quality attributes of RV frameworks because it facilitates expressivity as well as performance. In general, a single logical formalism is not sufficient for serving different use-cases [58]. Similar can be argued about the efficiency of monitoring algorithms as different variants of monitoring algorithms for a logical formalism are useful for different use-cases. Usability is also an important quality attribute because it is required that RV frameworks offer a user-friendly interface to users.

4.5.1 Evaluation of RitHM's Architecture

In this sub-section, we describe the ATAM evaluation for RitHM. As a first step in ATAM, we identify the business drivers for RitHM as follows.

[2] The evaluations are based on available description about architectures of the tools and frameworks.

- Provide a user-friendly interface to an RV system.
- Support tailoring RitHM for different use-cases w.r.t. monitoring different types of specifications.
- Support tailoring RitHM for different use-cases w.r.t. monitoring under constraints for efficiency in terms of execution-time, memory-usage, etc.
- Support RitHM with different types of reusable configurations on various OS platforms and allow integration of RitHM into diverse types of software systems and their test-beds.

Based upon the business drivers, we identified following quality attributes for RitHM's architecture.

- Modifiability: A user should be able to tailor a RitHM instance for different needs w.r.t. the requirement of using different logics for verification, and different constraints w.r.t. efficient monitoring in terms of resource-consumption.
- Usability: A user should be able to develop specifications and configure a RitHM instance in an easy way.
- Performance: RitHM should enable efficient monitoring in terms of execution time and memory usage.
- Modularity and Portability: A user should be able to use RitHM's modules within applications and their test-beds across different OS platforms.

We note that modifiability is one of the most important attributes of an RV setup because RV setups primarily perform the task of automated code-generation for the verification task under different use-cases for various domains. A requirement of synthesizing monitors in an automated manner for different types of specifications and software systems underscores the importance of modifiability for RV frameworks. Further, different use-cases for RV often show a trade-off between modifiability and performance. When an architectural decision can cause such trade-off, the core architecture of RitHM is designed to focus more on modifiability than performance in order to serve different use-cases. Due to a use of plugins, a user could develop efficient plugins for many use-cases.

We evaluated RitHM's component-based architecture for the aforementioned critical quality attributes. We depict some important findings of this evaluation. Architectural approaches for RitHM's design are as follows.

- Object-oriented and component-based architecture with most of the key processing components supported as plugins

- Layered architecture with segregation of front-end components from back-end components.

- A use of configuration files for setup of an RitHM instance.

- Integration of reflection features to load and utilize RitHM plugins at run time

Table 4.2 depicts various scenarios for the usage of RitHM. For the sake of brevity, we consider a few scenarios of RitHM, where programs or execution traces are monitored with a set of built-in plugins under minimal changes to RitHM's configuration. Use-case scenarios U1-U8 depict such cases of common use of RitHM. For example, monitoring of FOLTL formulas using the parallel algorithm is a typical use-case, which does not warrant changes to any of the components in RitHM's architecture. Such typical use-cases often identify 'performance' as the key quality attribute of RitHM. Various design alternatives such as a use of parallel-algorithms and heterogeneous back-ends highlight the provisions in RitHM's various plugins for achieving the desired performance requirements for respective use-cases. Such requirements differ for different use-cases. Additionally, different plugins for controlling invocation of monitors allow reducing the overhead of RV due to context switches.

We note some of the design decisions, which focus on 'performance' as the quality attribute. The associated non-risks are also described.

- The architectural decision of segregation of the Invocation-controller component and the Monitor component is motivated by the importance of performance in RV. This segregation allows that a user could configure a RitHM instance by using efficient monitoring algorithms as well as efficient means of controlling the invocation of the monitoring algorithms. The segregation of components in this case allows a fine-grained control to meet the efficiency requirement.

- The architectural decision of allowing interoperability between different *Monitor* plugins allows using efficient algorithms for different sub-specifications of a specification.

- A use of heterogeneous back-ends with parallel algorithms allows RitHM to efficiently monitor specifications.

The scenarios in Table 4.2 include three categories namely *use-cases*, *growth* and *exploratory* scenarios. The growth scenarios depict modifications to RitHM in which specific components of RitHM's architecture are changed. On the other hand, the exploratory scenarios depict extensive modifications to RitHM's components and their interactions. Growth scenarios G1-G12 highlight a need for *modifiability* as one of the quality attributes of RitHM's architecture. This attribute is in turn derived from a need for use-case specific performance and expressivity requirements. In order to address this quality attribute, we took following architectural decisions. The non-risks associated with the architectural decisions are also noted.

- RitHM's architecture has two main layers. RitHM's front-end components are decoupled from its back-end because the modifiability scenarios of the front-end components differ from those of the back-end components. This design decision also allows porting multiple front-end components in combination with a back-end component, and vice versa.

- Various components of RitHM's architecture are supported as plugins, and a built-in hierarchy of base classes is provided to facilitate the implementation of plugins.

- *Monitor* is separated from *Verdict-listener* allowing changes in *steering* actions without any changes in the Monitor.

- *Predicate-evaluator* is separated from *Observer* and *Monitor*. This allows changing predicate-evaluation scripts without modifications in Observer and Monitor.

- The separation of *Invocation-controller* from *Monitor* allows changing overhead control schemes without changing monitoring algorithms.

- The separation of *Observer* from other components allows changes in the process of event extraction without changing the remaining RV setup.

A need for portability is highlighted by Scenario G6 where a RitHM instance needs to deployed on Mac OS. Scenarios G7 and G8 highlight a need of portability between RitHM's front-end components and its back-end components. Further, scenarios G10 and G11 highlight a need for modularity, where RitHM's individual components need to be used in conjunction with other third-party tools. This gives flexibility to a user to independently use individual components of RitHM within an RV setup. For achieving portability and modularity, we made following architectural decisions.

- Implementation of RitHM's core components is done in Java, which provides an OS independent layer and allows RitHM to be ported on different OS platforms.
- RitHM's front-end components are separated from its back-end components, which allows porting different front-ends to back-ends and vice versa.
- A set of APIs is provided for each of RitHM's components, which are supported as plugins.

The scenarios U5, U7, G8, G9 and G12 show a need of usability as one of the important quality attributes. Writing specifications is one of the most important and complex phases of performing RV. Thus, easy and user-friendly ways of describing specifications are required for RV. In the context of programming languages, this has been done by (1) identifying commonly used patterns of usage of constructs (2) developing high-level constructs to abstract the complexity of such patterns. Similar process can prove beneficial in case of RV. For facilitating usability, we took following design decisions.

- Separate monitor-generation process from the specification-validation and translation phases. This allows *Specification-rewriter* to rewrite high-level specifications into low-level specifications, which can be inputted to *Monitor-synthesizer*.
- A use of configuration files allows modifying only required parts of an existing configuration of a RitHM instance. Further, web-interface can be developed as a presentation-layer, and it can create configuration files based upon a user's input. Such layered architecture can allow a further increase in the usability of RitHM by adding more features in the presentation layer.

The exploratory scenarios E1-E7 indicate a need for exhaustive and complex changes in specific contexts. Such scenarios depict a need to change RitHM's internal components. These scenarios involve multiple quality attributes, and they help to identify various sensitivity points and trade-off points.

The ATAM analysis of RitHM can be summarized in the form of *risks*, *non-risks*, *sensitivity-points* and *trade-off points*. Various non-risks have been described previously with the associated architectural decisions.

The trade-off points are described as follows.

- This decision of using Java for implementing RitHM marks a trade-off between performance and portability. While the performance gap between 'C' and Java is decreasing [72], such gap can still prove to be significant under various contexts. For example, embedded systems use ‘C’ code to achieve the specified performance under strict constraints for computing resources. The use of Java for the development of RitHM shows a trade-off point between performance and portability as Java is more portable than ‘C’ on different OS platforms.

- RitHM internally uses a hash map to store the details of an event, i.e., a program state. While this allows less modifications to RitHM’s Observer plugins for many of the use-cases, it reveals a trade-off between performance and modifiability because *application-specific* formats often tend to be more efficient in terms of memory and execution-time.
- Using dynamic buffers instead of static buffers provides flexibility to add different schemes for controlling the monitor-invocation with minimal coding changes. This decision also depicts a trade-off between performance and modifiability as static buffers can prove to be more performance efficient than dynamic buffers.
- A decision to use configuration files for setting up a RitHM instance shows a trade-off between usability and performance. While using configuration-files adds another layer and may slow-down the initial processing, it increases the usability by allowing a user to reuse previously created configurations.

The risks are described as follows.

- A support for multiple programming languages would require changes in existing plugins of following types: Invocation-controller, Predicate-evaluator, Verdict-listener and Monitor-generator.

The sensitivity points are described as follows.

- A provision to allow variable number of command-line arguments in RitHM’s command processor increases the modifiability of all RitHM components, and implementations of new plugins can utilize additional arguments.
- The task of developing monitor synthesizers for FPGAs, a unique back-end, poses challenges in terms of modifiability of most of the components of RitHM. As RitHM is designed to be a general-purpose framework, we note this sensitivity point during our evaluation.

- The segregation of front-end components from back-end components shows a sensitivity point in terms of usability, modifiability and modularity. This decision increases the ability of the architecture to exhibit these quality attributes.

- Providing APIs for *Observer* shows a sensitivity point for modifiability, and this design decision allows a use of various available instrumentation frameworks with RitHM.

4.5.2 Evaluation of RiTHM's Architecture

The architecture of previous version of RitHM, i.e., RiTHM has been described in [63]. As previously indicated, we evaluate the architecture based upon modifiability, performance and usability as the quality attributes. We report the findings in the form of non-risks, risks, sensitivity points and trade-off points for the architecture. The scenarios, considered for the evaluation, are a sub-set of scenarios considered for the evaluation of RitHM framework's architecture. The findings show that RiTHM's architecture focusses primarily on *performance*. However, the architecture requires improvements w.r.t. modifiability and usability because many of the components are tightly coupled and are not supported as plugins.

The non-risks are identified as follows.

- The design supports efficient methods for controlling the invocation of monitors by using time-triggered monitors with fixed as well as dynamic polling period. These schemes can be used to reduce the extra overhead of monitor invocation for different use-cases w.r.t. memory consumption and no. of context switches.

- The design supports utilizing GPUs for executing parallel monitoring algorithms for Ltl. This enables efficient monitoring in terms of execution time as well as CPU consumption.

The risks are identified as follows.

- *Monitor-generator* and *Instrumentor* are tightly coupled. The architecture can be improved by segregating the responsibilities of these components. With current design, monitoring can only be performed on programs, instrumented by RiTHM's instrumentor. Further, modifying the architecture for offline monitoring of traces involves significant changes in *Monitor-generator* because its current implementation only supports the data-format produced by the instrumentation.

Type	ID	Scenarios	Changes in RitHM's Components
Use-cases	U1	Monitor Ltl formulas with LTL$_3$ monitor on XML trace with ETRV	None
	U2	Monitor FOLTL formulas on CSV trace with BTRV the	None
	U3	Monitor MTL formulas with MTL monitor on CSV trace with BTRV and ‘Lua‘ script for predicate evaluation	None
	U4	Perform online monitoring of a Java program to monitor MTL formulas with RMTL monitor	None
	U5	Monitor LTL formulas using Dwyer's patterns with LTL$_3$ monitor on trace of a Java program.	None
	U6	Monitor PTLTL formulas on a C program using RitHM's server mode	None
	U7	Change formulas for an existing RitHM setup	None; Changes in RitHM's configuration file
	U8	Monitor FOLTL formulas on XML traces with parallel and sequential algorithms by dividing formulas into sub-formulas	None
Growth	G1	Add functionality to monitor a new logical formalism	New plugins for Monitor-synthesizer and Specification-rewriter
	G2	Add functionality to extract data in Common Log Format from a web-server	New Observer plugin
	G3	Add functionality to steer a Java Program upon violation of a safety formula	New Verdict-listener plugin
	G4	Add functionality for predicate-evalutor scripting in Scala	New Predicate-evaluator plugin
	G5	Add functionality for control-theoretic Invocation-controller implementation	New Invocation-controller plugin
	G6	Port RitHM on Mac OS	Migrate and test plugins which use native libraries; Execute smoke tests
	G7	Add implementation of a new monitoring algorithm for FOLTL	New Monitor-synthesizer plugin
	G8	Add specification-processing feature for patterns of Regular Expressions and PTLTL	New Specification-rewriter plugin which extends existing plugins for PTLTL and Regular Expressions
	G9	Use RitHM's Specification-rewriter for Dwyer's pattern with other tool providing implementation of LTL monitoring algorithm	None; Changes in scripts
	G10	Use RitHM's Verdict-listener for other monitoring tools	None; Changes in scripts
	G11	Add new command-line options for controlling monitor invocation	Changes in Invocation-controller plugin
	G12	Develop high-level language based on patterns for MTL	New Specification-rewriter plugin
Exploratory	E1	Change RitHM's data-structures for storing events, predicates and verdicts	Major changes required in Observer, Monitor-synthesizer, Monitor Predicate-evaluator and Invocation-controller plugins; Changes in class-hierarchy for monitor's verdicts
	E2	Add FPGA back-end to RitHM	Major changes required in Observer, Monitor-synthesizer Predicate-evaluator, Invocation-controller and Verdict-listener plugins
	E3	Change memory allocation scheme for RitHM's event-buffer from dynamic to static	Major changes required in Observer, Monitor and Invocation-controller plugins
	E4	Change base APIs for Monitor-synthesizer, Specification-rewriter, Observer, etc.	Same as that of E1
	E5	Add distributed monitoring to RitHM	Same as that of E1
	E6	Change RitHM's command-line parameters	Same as that of E1 and the Command-handler
	E7	Add new component to RitHM's design for searching specifications with patterns	Changes in front-end components

Table 4.2: Different Scenarios for Usage of RitHM

- There is no separate *Observer* component in the architecture. An addition of this component to the architecture can improve the modifiability. With current design, a use of the tool for online monitoring via APIs requires significant changes in order to ensure that different data-formats can be consumed by generated monitors. If a new instrumentation utility needs to be utilized, then similar changes are required in various components.

- Since the design does not support the front-end components as plugins, significant changes are needed in order to process different logical formalisms. This also affects *usability* because the specification-processing and monitor-generation phases are performed by *Monitor-generator*.

- *Globalizer*, which changes the storage class of certain variables to 'global' for instrumentation purposes, may cause problems for large programs because global variables are generally not recommended. Such transformation can affect modularity as well as readability of programs.

- The verdict of a monitor is logged in a file. A separate component for steering a program based upon the monitor's verdict is not supported.

The sensitivity points are identified as follows.

- RiTHM uses a configuration file for the setup of a RiTHM instance. Any changes in the syntax of configuration files are likely to cause modifications in multiple components.

- A use of *Globalizer* and *Glue Code Generators* makes the architecture sensitive to changes because a major change in these components is likely to affect most other components including *Monitor-generator* and *Controller-generator*.

- Using a third-party tool, i.e., `ltl3tools` for monitor generation causes the architecture to be sensitive to changes in the third-party tool[3].

The tradeoff points are identified as follows.

- A program and the corresponding monitor run as different threads in a single process. While this reduces the runtime overhead, it affects the modifiability as the two components are tightly coupled.

[3] This was observed during our work on porting RiTHM to Mac OS.

- While changing the storage class to 'global' during the instrumentation reduces the runtime overhead in terms of memory usage, it affects the modifiability of the program. Further, the usability of the setup also gets affected because a use of global variables is generally not recommended.

4.5.3 Evaluation of MOP's Architecture

The architecture of MOP framework has been described in detail in [58]. As previously indicated, we evaluate the architecture based upon modifiability, performance and usability as the quality attributes, and we report the findings in the form of non-risks, risks, sensitivity points and trade-off points for the architecture. The findings highlight that MOP is a highly modifiable and modular framework. However, it can be further improved in terms of its features for facilitating efficient monitoring.

The non-risks are identified as follows.

- A use of *language clients*, which allow monitor-generation for different programming languages, provides an easy way to add monitor-generation for softwares written in different programming languages.
- *Logic-plugins* facilitate adding monitor-generation process for a new logical formalism.
- A use of language-independent pseudocode as a generic way of synthesizing monitoring procedures allows MOP to add support for a programming language without modifying its logic-plugins.
- *Recovery-handlers* enable a program to be steered into a correct state, and such handlers can be provided as code snippets in respective programming languages.
- *Logic-plugins* can also facilitate the process of creating extensions and high-level logical formalisms of existing logical formalisms.
- *Language clients* also facilitate utilizing heterogeneous back-ends for efficient execution of monitor.

The risks are identified as follows.

- In MOP, a *language client* handles the instrumentation task in one of its layers. As described earlier, different instrumentation tools and frameworks have been developed for efficient instrumentation techniques. Thus, performance improvement can be achieved by adding a standard interface for using different instrumentors as plugins.
- An addition of component in the design to use various overhead-control schemes such as TTRV and BTRV can help to reduce the runtime overhead.
- The standard syntax for configuring *language clients* may be restrictive for evaluating predicates (resp. propositions) for various use-cases of monitoring.

The sensitivity points are identified as follows.

- A use of standard syntax by language clients of MOP framework highlights a sensitivity point. Any modifications to this syntax would cause changes in all layers of the respective language client.
- A use of pseudocode for monitor-generation shows a sensitivity point, where changes in the constructs of the pseudocode affect most of the components of the architecture. Thus, modifications in the constructs of the pseudocode would result in modification and validation of the functionality of most other components.

The tradeoff points are identified as follows.

- A use of pseudocode for monitor-generation shows a tradeoff point between performance and modifiability. Using pseudocode facilitates modifiability. However, it may affect the performance because a translation of some pseudocode constructs to specific programming languages may not be available. Hence, some efficient monitoring algorithms for specific fragments of logical formalisms cannot be used in certain programming languages. For example, pseudocode constructs for parallel monitoring may not get used in a programming language, which does not provide the corresponding parallelism requirements.

4.5.4 Evaluation of Larva's Architecture

Larva is a tool for monitoring real-time properties of Java programs. A brief description of Larva's architecture is available in [22]. The findings show that Larva's architecture is modifiable, and certain improvements can be made in the architecture to increase the efficiency of monitor execution and the usability of the front-end components.

The non-risks are identified as follows.

- Larva supports multiple logics, which are translated to dynamic automata with timers and event (DATE). With this feature, it is possible to develop high-level extensions of the logics without changing the monitor synthesis algorithm.
- The property-analysis feature allows an estimation of extra overhead caused due to monitoring.

The risks are identified as follows.

- Larva only supports AspectJ based instrumentation. Thus, processing traces for offline monitoring and monitoring applications using other instrumentation tools would require significant changes in most of the components.
- The architecture does not support porting different back-ends to front-ends, and vice versa. Thus, different monitoring algorithms cannot be used for specifications described using a logical formalism.
- Although extra-overhead can be measured by using property-analysis feature, the architecture does not support components for controlling the overhead due to monitor invocation.
- The *feedback* system relies on AspectJ. Thus, any changes in instrumentation scheme require changes in feedback system as well.

The sensitivity points are identified as follows.

- The architecture heavily relies on AspectJ technique. Any change in this instrumentation scheme would cause changes in components handling feedback, evaluation of predicates, etc.
- While DATE is a highly expressive logical formalism, a high-level specification language ,which can be translated into a automata can increase the usability. Such high-level languages are useful because automata construction processes are generally complex.

- A decision of using DATE as a ‘native’ logical formalism for front-end enables Larva to support other logical formalisms such as Lustre, QDDC and DC.

There were no tradeoff points identified based upon the given description of the architecture.

4.5.5 Discussion

Following conclusions can be drawn based upon the observations of the ATAM evaluation for the aforementioned tools and frameworks.

- Although architectures of some tools such as MOP and Larva exhibit features for facilitating usability, RitHM’s business drivers directly identify usability as one of the primary quality attributes.
- Most of the architectures primarily focus on one of the quality attributes and need improvements w.r.t. other quality attributes. RitHM’s architecture focuses on multiple quality attributes and attempts to achieve a balanced tradeoff between them.
- Only RitHM’s architecture exhibits a clear segregation of responsibilities between front-end and back-end components of an RV setup.
- While most of the components of the architectures of RiTHM, MOP and Larva are sensitive to some architectural decisions, RitHM’s architecture tries to limit such sensitivities, which affect a large number of components of an RV setup. RiTHM’s architecture is sensitive w.r.t. the modifications in its instrumentation process. For MOP, the architecture is sensitive to modifications in the syntax of *language clients* and the pseudocode. Larva relies on its usage of aspects in most of its components. For RitHM, we tried to limit such sensitivities by making various architectural decisions.

Thus, the analysis shows that RitHM’s business drivers and the unique features of its architecture are important to simultaneously achieve the various goals of an RV setup.

Chapter 5

Case-studies for Monitoring using RitHM

In this Chapter, we present the results of an evaluation of RitHM's features using real-world data-sets. The first case study depicts our evaluation of monitoring Google cluster-usage traces using RitHM. Further, the second case study shows the applicability of RitHM's features while monitoring the traces of an Engine Control Unit (ECU) of a Toyota 2JZ engine.

5.1 Monitoring Google Cluster Traces

A Google cluster is a set of machines, packed into racks, and connected by a high-bandwidth cluster network [68, 78]. Work is performed on the cluster in the form of jobs, each comprising one or more tasks. Tasks and jobs are scheduled onto different machines. The data-set used in this case-study, provides data from an 12.5k-machine cell over about a month-long period in May 2011. Tasks and jobs progress through different states such as 'submitted', 'scheduled', 'failed', etc. We use a fragment of first-order LTL to define and verify following properties for multiple jobs and tasks. These properties verify the behavior of jobs and tasks w.r.t. their transitions.

Property 1: $(\forall task \cdot submit(task) \rightarrow \Diamond finish(task)) \wedge (\forall job \cdot submit(job) \rightarrow \Diamond finish(job))$. This property states that for all jobs and tasks, a 'submit' event is eventually followed by a 'finish' event.

Property 2: $(\forall task \cdot upd\_running(task) \rightarrow upd\_running(task)\,\mathbf{U}\;\; finish(task)) \wedge (\forall job \cdot$

No.	**No. of Tasks**	**No. of Jobs**
1	4,000,000	95,384
2	5,000,000	129,456
3	6,000,000	162,066
4	7,000,000	188,711
5	8,000,000	210,350

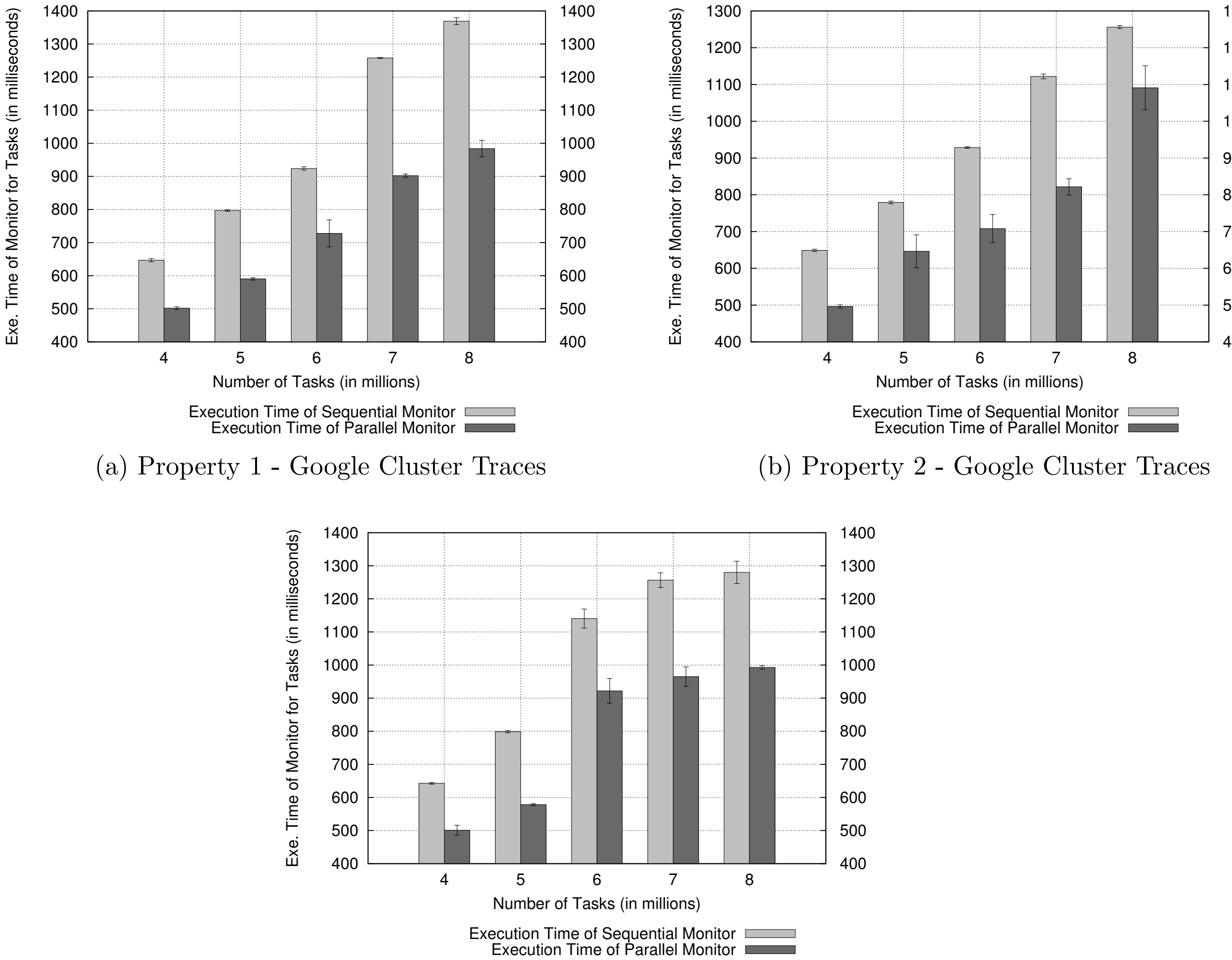
Table 5.1: No. of Tasks and Jobs

$upd\_running(job) \rightarrow upd\_running(job)\,\mathbf{U}\,finish(job))$. This property states that for all jobs and tasks, if an update happens at runtime, then it would keep happening until the job or the tasks finishes.
Property 3: $(\forall task \cdot submit(task) \rightarrow \bigcirc\,(upd\_pending(task) \vee schedule(task) \vee kill(task) \vee fail(task) \vee evict(task)))\wedge$
$(\forall job \cdot submit(job) \rightarrow \bigcirc\,(upd\_pending(job) \vee schedule(job)\vee kill(job) \vee fail(job) \vee evict(job)))$. This property states that after a job or a task is submitted, it fails or is updated or is scheduled or is killed or is evicted.

We monitored these properties for jobs and tasks for different configurations. For each configuration, a certain number of tasks and corresponding jobs are selected. Our experimental environment for this experiment is identical to the one used in the previous chapter. The number of tasks and jobs used for different configurations in this experiment is described in Table 5.1. We note that more number of tasks may exist for the specified jobs. For the sake of simplicity during the experiment, we chose the given number of tasks and the corresponding jobs.

We utilized a variant of the algorithm [56] developed by Medhat et al. for monitoring the properties. We decompose the property into sub-properties for the jobs and the tasks. The sub-properties are monitored separately. We also note that the traces for jobs and tasks are produced separately. With reference to our findings in the previous chapter, we utilize the parallel algorithm for monitoring the property for the tasks. However, we utilize the sequential algorithm for monitoring the property for the jobs because a use of the parallel algorithm is unlikely to achieve the benefits considering the given no. of jobs. In Figure 5.1, the execution time for performing the aggregation operation for the evaluation of the quantifiers is reported for the sub-specificaton for monitoring the *tasks*. For monitoring the *jobs*, this execution time for performing aggregation is found to be identical because parallelization is not enabled for both variants.

We note that the interoperability of parallel and sequential monitors for the same fragment of first-order Ltl provides the flexibility to utilize both sequential and parallel monitoring algorithms. In this case, the property was divided into parts and each of these parts are monitored using variants of the monitoring algorithm. For large-scale data, the utilization of multiple back-ends along with the interoperability feature can prove beneficial.



(a) Property 1 - Google Cluster Traces

(b) Property 2 - Google Cluster Traces

(c) Property 3 - Google Cluster Traces

Figure 5.1: Comparison of Monitoring Overhead for Google Cluster Traces

5.2 Monitoring Engine Control Unit (ECU) Traces

In this case study, we analyze traces [57] from an ECU, which controls multiple actuators and sensors for a Toyota 2JZ engine. The traces depict the metrics of various sensors of the ECU during a step test for a duration of 34 seconds. The ECU controls fuel injection based upon the values of parameters namely 'Engine RPM' and 'Engine Load'. In general, the value of parameter 'Engine Load' can be measured by multiple ways such as throttle position and manifold pressure. The lambda value is the air-to-fuel ratio, which depends upon the fuel injection process. In general, if the lambda value is over 1.0, then the mixture of air and fuel is considered 'lean'. If the lambda sensor reports value less than 1.0, then the mixture is considered 'rich'. While 'lean' mixtures may cause engine damage due to increased temperatures, 'rich' mixtures result in low mileage. A lambda value of 1.0 is considered ideal. Based on these parameters, we define properties for the verification of ECU traces. The values of the aforementioned parameters for the dataset are shown in Figure 5.2 using the plots. The properties define conditions over the value of the lambda sensor w.r.t. the values of 'Engine RPM' and 'Engine Load'.

Let *rpm_over_4000* be an atomic proposition, which marks the condition when the value of 'Engine RPM' goes over 4,000 from a value less than 4,000. Similarly, *rpm_below_4000* be a proposition, which marks the condition when the value of 'Engine RPM' goes below or becomes equal to 4,000 from a value greater than 4,000. In similar way, other propositions are defined for different values of 'Engine RPM', lambda sensor and 'Engine Load'. We also refer to the parameter 'Engine Load' as load point. Some atomic propositions are defined using comparison operators over the values of the aforementioned parameters.

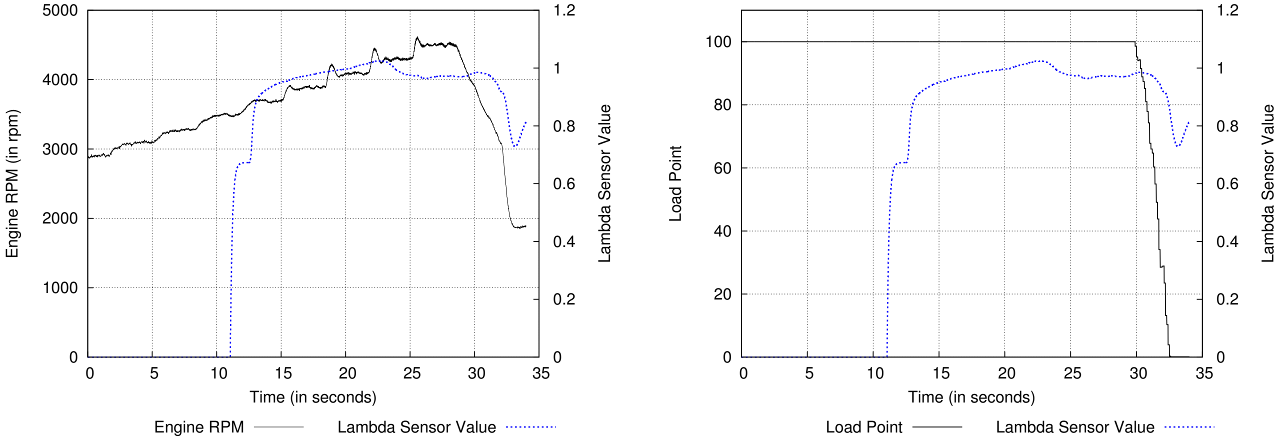
Using Dwyer's patterns [28, 29], we define and monitor the properties shown in Table 5.2. Our experimental environment for this case-study is identical to the one used in the previous case-study. The described properties belong to the classes namely 'Response', 'Absence', 'Existence' and 'Universality. We implemented a *Specification-rewriter* plugin for a subset of Dwyer's patterns, and it translates the patterns into corresponding LTL formulas, which are used to synthesize the corresponding monitors. The first property states that *after* the value of 'Engine RPM' goes below 4,000 it is always a case that when the value of 'Load point' becomes less than 80, then *eventually* the value of the lambda sensor is in the range of 0.8 and 1.0. This property belongss to the 'Response' class. The second property states that after the value of 'Engine RPM' goes over 4,000, the lambda value goes above 1 until the value of 'Engine RPM' goes below 4000. This property belongs to the 'Existence' class [28]. The third property states that when the value of 'Engine RPM' is above 4000, the lambda value is in the range of 0.8 and 1.2. The third property belongs to the 'Universality' class. The fourth property is similar to the third property. It indicates

No.	**Formula Using** **Dwyer's Pattern**	**Equivalent** **LTL Formula**	**Size of** **LTL Formula**
1	$\underline{\text{After}}\ \mathit{rpm\_below\_4000}$, event $0.8 \le \lambda \le 1.0$ $\underline{\text{Responds}}$ to event $\mathit{load\_pt} < 80$	$\Box(\mathit{rpm\_below}\_4000 \rightarrow$ $\Box(\mathit{load\_pt} < 80 \rightarrow$ $\Diamond\, 0.8 \le \lambda \le 1.0))$	8
2	$\underline{\text{After}}\ \mathit{rpm\_over\_4000}$ $\underline{\text{Until}}\ \mathit{rpm\_below\_4000}$ $\lambda \ge 1$ $\underline{\text{becomes}}$ true	$\Box((\mathit{rpm\_over}\_4000 \wedge \neg \mathit{rpm\_below}\_4000)$ $\rightarrow$ $(\neg(\mathit{rpm\_below}\_4000)\, \mathbf{U}\, ((\lambda \ge 1) \wedge \neg(\mathit{rpm\_below}\_4000))))$	9
3	$\underline{\text{Between}}\ \mathit{rpm\_over\_4000}$ and $\mathit{rpm\_below\_4000}$, $0.8 \le \lambda \le 1.2$	$\Box((\mathit{rpm\_over}\_4000 \wedge \neg \mathit{rpm\_below}\_4000 \wedge$ $\Diamond \mathit{rpm\_below}\_4000) \rightarrow (0.8 \le \lambda \le 1.2$ $\mathbf{U}\, \mathit{rpm\_below}\_4000))$	10
4	$\underline{\text{Between}}\ \mathit{rpm\_over\_2000}$ and $\mathit{rpm\_below\_2000}$ $\lambda \not\ge 1.2$	$\Box((\mathit{rpm\_over}\_2000 \wedge \neg \mathit{rpm\_below}\_2000$ $\wedge \Diamond \mathit{rpm\_below}\_2000) \rightarrow$ $(\neg(\lambda \ge 1.2)\, \mathbf{U}\, \mathit{rpm\_below}\_2000))$	11
5	$\underline{\text{Between}}$ $(\mathit{rpm\_below\_4000} \wedge \mathit{load\_pt} < \mathit{80})$ and $(\mathit{rpm\_below\_2000} \wedge \mathit{load\_pt} < \mathit{80})$ $\lambda \ge 0.8$ $\underline{\text{Responds}}$ to $\lambda < 0.8$	$\Box(((\mathit{pm\_below}\_4000 \wedge \mathit{load\_pt} < 80) \wedge$ $\neg((\mathit{rpm\_below}\_2000 \wedge \mathit{load\_pt} < 80)) \wedge$ $\Diamond((\mathit{rpm\_below}\_2000 \wedge \mathit{load\_pt} < 80))) \rightarrow$ $(\lambda < 0.8 \rightarrow (\neg(\mathit{rpm\_below}\_2000 \wedge \mathit{load\_pt} < 80)\mathbf{U}$ $(\lambda \ge 0.8 \wedge \neg(\mathit{rpm\_below}\_2000 \wedge \mathit{load\_pt} < 80))))$ $\mathbf{U}(\mathit{rpm\_below}\_2000 \wedge \mathit{load\_pt} < 80))$	14

Table 5.2: Ltl Properties for ECU verification

an absence of ‘lean’ mixture with lambda value over 1.2, when the value of ‘Engine RPM’ is over 2000. The fourth property belongs to the ‘Absence’ class. The fifth property, which belongs to the ‘Response’ class, states that when the value of ‘Engine RPM’ is in the range of 4,000 and 2,000 and the value of ’Engine load’ is less than 80, if lambda value goes below 0.8, then it eventually becomes greater than or equal to 0.8.

Table 5.2 shows the size of the corresponding Ltl formulas. The size of a Ltl formula indicates its complexity. As shown in the table, Dywer's patterns can be used to develop high-level specifications, and they help to abstract the complexity of the underlying Ltl specifications. For the third property in Table 5.2, the Ltl formula's size is 10. However, the property can be expressed using the pattern ‘P is true $\underline{\text{between}}$ Q and R’, which uses the ‘between’ operator. As shown in Table 5.2, an application of such high-level operators ensures that even though the complexity of the Ltl formulas increases, the equivalent specifications, which use such operators, are less complex. Thus, such high-level operators abstract the complexity of the corresponding Ltl formulas. Consequently, an ability to process high-level specifications increases the *usability* of an RV framework. Further, these findings also underscore the importance of RitHM's design feature, which segregates the front-end components from the back-end components in addition to the provision of the functionality for rewriting specifications.



(a) Lambda value and Engine RPM

(b) Lambda value and Load Point

Figure 5.2: Metrics of Engine Control Unit (ECU)

5.3 Lessons Learned

The implementation of RitHM helped us to understand various nuances of an RV setup. In particular, the development of a comprehensive set of APIs for RV was a challenge. Our participation in CSRV'14 helped the understanding of various use-cases and important aspects about the comparative evaluation of RV tools. The evaluation parameters of the competition helped to understand important components of an RV set-up.

We found that the current state of RV techniques requires a user to posses in-depth knowledge of different logical formalisms. Further, our interaction with the personnel in software development industry revealed that the developers and testers often prefer simple ways of describing a system's specifications. The logical formalisms of RV techniques have evolved from those of other formal verification techniques such as model checking. Model checking is performed by experts. Consequently, complex logical formalisms are used for specifying a system's specifications. On the contrary, runtime verification, which is a *lightweight* but *scalable* formal method, aims to address the limitations of formal verification techniques, and it is aimed to automate the process of verification for various types of practitioners. This suggests that RV frameworks need to support easy ways of describing specifications. RitHM provides support for extending its features using plugins to facilitate the development of high-level abstractions of existing logical formalisms.

Further, we note that a use of heterogeneous back-ends such as GPU or multi-core

CPU requires a profiling effort in order to tune a monitor for achieving the predicted efficiency. This profiling effort can prove prohibitive for the scalability of RV techniques as it is intended to be a generic method to synthesize and execute the monitors in an *automated* manner. We believe that the advancements in the usage of GPUs and FPGAs for general-purpose computing would help to counter this challenge.

Additionally, we noticed that a use of parallel algorithms can be useful for monitoring parametric specifications. In general, runtime verification is a 'Memory Bound' workload [25] and not a 'Compute Bound' workload, i.e., monitors generally do not perform computationally intensive operations. Thus, an effective use of accelerators such as GPUs would require efficient ways of memory transfer. We believe that the advancements in the usage of GPUs for general-purpose computing would help to counter this challenge.

Chapter 6

Problem Description for Loss-tolerant Monitoring

In this chapter, we formally define the problem of monitoring of Ltl in presence of a transient loss of events in an execution trace. Further, we discuss the monitorability of Ltl under a transient loss of events and describe the criteria for the same.

6.1 Our Model

As shown in Figure 6.1, we assume that the program P is run under the supervision of an *observer* O, which incrementally extracts events from P and feeds them to the monitor $\mathcal{M}$, which then verifies whether $[\sigma \models \varphi]_{RV}$ [13]. Due to the lossy link between the observer O and the program P, the extracted word may contain gaps. We assume that the observer transmits the trace to the monitor without any extra loss. We also assume that the execution trace exhibits a transient loss of events.

Definition 7 (Transient Loss)**.** *Let $\sigma = a_1, a_2, \ldots, a_n \in \Sigma^*$ denotes a finite trace of size n. The trace σ exhibits a transient loss of a sequence of events iff there exists a non-empty*

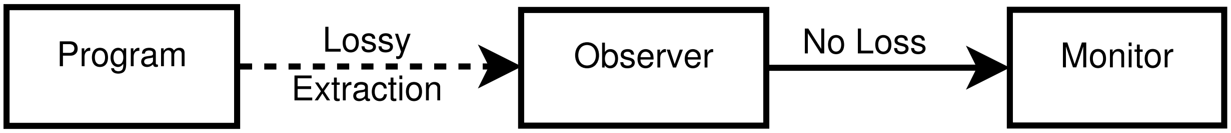


Figure 6.1: System Model

and finite set of bounded integer intervals, $I_s = \{[i_1, j_1], [i_2, j_2], \cdots, [i_m, j_m]\}$ with $m < n$, such that $\forall I \in I_s, \forall k \in I, a_k = \chi$. Here, χ denotes an unknown element (state) in the trace σ.

Formally, a transient loss is a finite loss of a sequence of events in an execution trace such that the loss is not permanent, and it is guaranteed that the observer can observe subsequent events in the trace. In the rest of the thesis, we assume that the alphabet is augmented with the unknown state χ, i.e, $\Sigma = \Sigma \cup \chi$. Our further findings consider aforementioned considerations about the model.

6.2 Formal Problem Description

Given a program P and an Ltl formula [65] φ, our goal is to identify whether φ can be *soundly* monitored at run time in presence of a *transient loss* of a sequence of events. In general, the verdict of a monitor is unknown for the part of an execution trace which cannot be observed. So, it entails that the system recovers from the transient loss, and the monitor is able to observe subsequent events in the trace. By soundness, we mean that in case it is possible to recover the events lost due to transient loss, the truth-value output by a monitor on a lossy trace would align with the truth-value of a monitor, which processes the corresponding complete execution trace. We note that when a monitor processes a lossy execution trace, it cannot produce a verdict for a prefix of the trace, when the prefix ends with the unknown part of the trace.

Monitoring some properties, such as $\Box\,(\textit{request} \rightarrow \Diamond\ \textit{response})$, in presence of a transient loss requires that the part of the execution trace, which is observed after the end of the loss, exhibits certain conditions. Such conditions require that after the loss, the subsequent part of the trace contains at least one input element, which belongs to $\Sigma \setminus \chi$. We formalize this using the concept of *loss-tolerant alphabet*, which is defined as follows:

Definition 8 (Loss-tolerant alphabet)**.** *A loss-tolerant alphabet, Σ_{-}, is a non-empty subset of $\Sigma \setminus \chi$.*

Similarly, to monitor properties such as $p \rightarrow \Box\Diamond\,(q\,U\,r)$ in presence of a transient loss, it is required that the state of the monitor, prior to the beginning of the loss, belongs to a pre-identified sub-set of states. We illustrate this using the concept of *loss-tolerant cluster*, which is defined as follows:

Definition 9 (Loss-tolerant Cluster)**.** *A loss-tolerant cluster is a non-empty set denoted by Q_s, which is a subset of Q, the set of all possible states of an RV-LTL monitor as given in Definition 6.*

Multiple combinations of loss-tolerant clusters and loss-tolerant alphabets may exist for an LTL formula. Further, there is a one-to-one relationship between the elements of loss-tolerant clusters and the elements of loss-tolerant alphabets for an LTL formula. During our experiments on the commonly used patterns [29] of LTL formulas, we found that the number of such combinations are few. Consequently, the constructed monitors incur minimal extra-overhead as later demonstrated by our empirical evaluation of the proposed method.

In later sections, we argue and prove that an execution-trace σ can be monitored in presence of transient loss iff the state of the monitor immediately before the beginning of a transient loss belongs to one of the loss-tolerant clusters, and *after* the end of the transient loss, the subsequent execution trace contains at least one input element, s, such that s belongs to the corresponding Σ_{-}. Although it may appear that the monitorability of an LTL formula strictly depends upon run-time conditions, our experiments show that some formulas can be monitored irrespective of any run-time conditions, because there exists a single combination of a loss-tolerant cluster and a loss-tolerant alphabet. Further, all the states of such formulas belong to a loss-tolerant cluster, and the corresponding loss-tolerant alphabet is equal to $\Sigma \setminus \chi$. An example of such formulas is $\Box \Diamond (a \, U \, b)$.

Formally, our problem can be stated as follows: Given an LTL formula φ and an execution trace σ, which exhibits a transient-loss of events, our goal is to

- Decide whether φ can be *soundly* monitored on σ.
- If yes, then construct a loss-tolerant monitor $\mathcal{M}^{\varphi}$ for the formula φ such that $[\sigma \models \varphi]_{RV} = \lambda(\delta(q_0, \sigma))$
- Identify the set of possible loss-tolerant clusters and corresponding loss-tolerant alphabets of φ. Elements of such sets are tuples, with each tuple being of the form (Q_s,Σ_{-}), where Q_s is a loss-tolerant cluster, and Σ_{-} is the corresponding loss-tolerant alphabet.

6.3 Monitorability of LTL formula under Transient Loss of Events

This section uses few examples to provide basic details about the criteria that identifies whether an LTL formula can be monitored in presence of a transient loss. The criteria is elaborated using the concepts of loss-tolerant alphabet and loss-tolerant cluster.

6.3.1 Motivating Examples

As per our model, a monitor $\mathcal{M}$, which is an FSM, consumes an element of set Σ, which is obtained from the Observer, O, and the monitor produces an output in the truth-domain $\mathbb{B}_5 = \{\top, \top_p, ?, \bot_p, \bot\}$, which is defined by a function $\lambda : Q \to \mathbb{B}_5$, where Q is the set of states of the monitor.

The formulas in the *obligation* [18] class can be satisfied or violated over a finite prefix of the execution trace. Under a transient loss, the monitor may not be able to observe some events in such finite prefix. Thus, as per our model, the formulas in the obligation class cannot be generally monitored unless a recovery of the lost events is possible. For example, for a safety formula, a transient loss may cause a monitor to miss an event, which causes a violation of the property, but the monitor cannot report a violation unless the lost event is recovered. Similar can be argued for a guarantee formula. However, in some trivial cases, the satisfaction or violation status of formulas in the *obligation* class can be determined in spite of a transient loss of the execution trace. For example, let us consider an LTL formula $\varphi_1 = \Box p$, where p is an atomic proposition. If φ_1 is already violated by a finite prefix of the execution trace, then a transient loss of subsequent trace will not change the verdict.

Further, the satisfaction or violation of certain LTL formulas cannot be determined with a finite prefix of an execution trace. For example, let $\varphi_2 = p \to \Box\Diamond(q\,\mathbf{U}\,r)$ be an LTL formula. Here, p, q and r are atomic propositions. An RV-LTL [13] monitor for this formula is as shown in Figure 6.2. Once the monitor is in state q_1 or q_2, the monitor can never deliver its verdict as $\top$ or $\bot$, because it can never be satisfied or violated by a finite word. However, as per the definition of monitorability [31], this formula is monitorable using RV-LTL semantics, because the verdict of an RV-LTL monitor for this formula differs for good and bad traces. In this case, the monitor delivers a verdict as $\top_p$, i.e., *'presumably satisfied'*, when a finite word ends with an element in which proposition r is *true*. Otherwise, its verdict is $\bot_p$, i.e., *'presumably violated'*. This monitor can perform monitoring in a sound manner in presence of a transient loss, because even though the

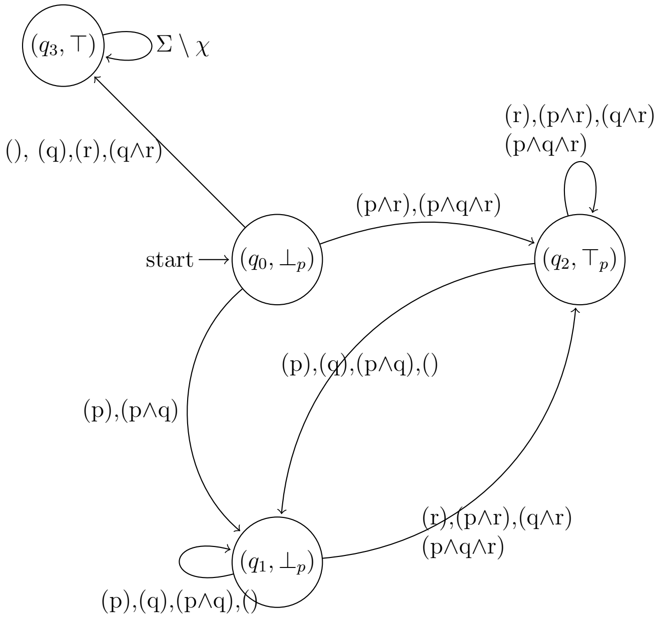


Figure 6.2: RV-LTL Monitor for $\varphi_2 = p \rightarrow \Box\Diamond\,(q\,\mathbf{U}\,r)$

current state of the monitor is unknown, the monitor can still provide a correct verdict for the next element in the trace provided its state immediately before the beginning of the transient loss is either q_1 or q_2. Thus, for this formula, the set of loss-tolerant clusters consists of a single cluster $Q_s = \{q_1, q_2\}$ and the loss-tolerant alphabet $\Sigma_- = \Sigma$. The states in Q_s upon receiving an identical element from Σ_- as the input yield a transition to the same next state. Although the previous state of the monitor is unknown due to a transient loss, the monitor can reach a correct next state. Until the interruption in observing the events is over, an RV-LTL monitor's verdict is unknown for the formula φ_2, because the input element or the current state of the monitor is unknown. However, once the monitor observes the subsequent events, it can determine a verdict, which is same as that of the monitor, which processes the complete execution trace.

We now consider another formula $\varphi_3 = \Box\,(p \rightarrow \Diamond\,q)$, whose RV-LTL monitor is as shown in Figure 6.3. This monitor can deliver a sound verdict for an execution trace with transient loss provided the part of the trace that is observed after the end of the loss, contains at least one input element $\sigma_i |\, \sigma_i \in \{p,\, q,\, p \wedge q\}$. Here, the loss-tolerant alphabet is $\Sigma_- = \{p,\, q,\, p \wedge q\}$. The loss-tolerant alphabet, for instance, does not include the subset $\{\neg p \wedge \neg q\}$. The existence of at least one element in Σ_- ensures that the next state of the monitor can be precisely determined.

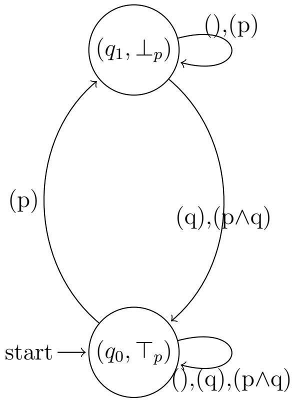


Figure 6.3: RV-LTL Monitor for $\varphi_3 = \Box\,(p \rightarrow \Diamond\, q)$

6.3.2 Formal Definition

Here, we formally define the monitorability criteria. Let φ be an LTL formula, and let $\mathcal{M} = (\Sigma, Q, q_0, \delta, \lambda)$ be the corresponding RV-LTL monitor for φ as described in Definition 6. Let $\sigma \in \Sigma^*$ denote a finite trace, which exhibits a transient loss. Let Q_s be a set of states of $\mathcal{M}$ such that $Q_s \subseteq Q$. Let Q_f be the set of final states of the *monitor*.

Definition 10 (Monitorability with Transient Loss)**.** *A formula φ can be monitored in presence of a transient loss iff:*

$$\begin{aligned}\exists \Sigma_{-}.\, \exists Q_s.\, \forall \alpha \in \Sigma.\, \forall (q_i, q_j) \,|\, q_i \in Q_s \wedge q_j \in Q_s.\\ ((\alpha \in \Sigma_{-}) \rightarrow (\delta(q_i, \alpha) = \delta(q_j, \alpha))) \wedge\\ ((\delta(q_i, \alpha) \in Q_s) \wedge (\delta(q_j, \alpha) \in Q_s)\\ \wedge(\delta(q_i, \alpha) \notin Q_f) \wedge (\delta(q_j, \alpha) \notin Q_f))\end{aligned}$$

The loss-tolerant cluster, Q_s, allows sound monitoring in presence of a transient loss of a sequence of events due to certain conditions on the transition function of the states in Q_s. These conditions ensure that although the monitor cannot make a transition during the period of the transient loss, it can still reach a state, when it observes the subsequent events, and this state is same as that of the corresponding RV-LTL monitor, which processes the execution trace without a loss.

All the states in a loss-tolerant cluster make transitions to the same next state within the same loss-tolerant cluster, when the states process an identical input element in Σ_{-}. Moreover, if a loss happens when the then current state of the monitor belongs to a loss-tolerant cluster, then the monitor cannot observe the events, and its state during such loss is unknown. However, the transitions of the loss-tolerant cluster ensure that the 'unknown' state would still be one of the states in the same loss-tolerant cluster. Once the interruption in the observation of events is over, the monitor can process the subsequent events, and it is ensured that the monitor can precisely determine its current state iff it can observe and process at least one element in Σ_{-}.

For some LTL formulas, it may be the case that there exists a single combination of a loss-tolerant cluster and the corresponding loss-tolerant alphabet such that $Q_s = Q$, and $\Sigma_{-} = \Sigma \setminus \chi$, i.e., the loss-tolerant cluster is equal to the set of all the states of the RV-LTL monitor, and the loss-tolerant alphabet is $\Sigma \setminus \chi$, the alphabet used by the RV-LTL monitor. $\Box \Diamond (a\, U\, b)$ is a formula, which exhibits the aforementioned conditions.

Chapter 7

Monitoring Lossy Traces

This chapter describes our monitor construction method and an algorithm to construct a loss-tolerant monitor, when the supplied LTL formula is determined to be monitorable in presence of a transient loss. Additionally, we provide a proof of correctness for the monitors constructed using our algorithm. Further, we provide the details of performance evaluation of our monitor construction algorithm.

7.1 Synthesis of Loss-tolerant Monitors

Algorithm 1 describes a process to find whether an LTL formula φ is monitorable in presence of a transient loss of events. FINDMONITORABLITY is the main procedure, which takes an LTL formula φ as input. If φ is monitorable, then it outputs $\top$. Otherwise, it outputs $\bot$. If φ is found to be monitorable in presence of a transient loss of events, then the procedure also synthesizes a loss-tolerant monitor. Otherwise, it returns the corresponding RV-LTL monitor.

As shown at Line 3, FINDMONITORABLITY constructs an RV-LTL monitor $\mathcal{M}^{\varphi}$ for φ using the method described by Bauer et al. [13]. From Line 4 to Line 17, it checks $\mathcal{M}^{\varphi}$ for the existence of possible combinations of a loss-tolerant cluster and a loss-tolerant alphabet that meet the criteria described in Definition 10. These combinations are stored in a list in the ascending order of the size of the loss-tolerant cluster at Line 14. All potential loss-tolerant clusters containing at least two states are verified. The reason being that the loss-tolerant clusters, which contain a single state and meet the criteria in Definition 10 are not important, because such subsets only contain the final states of an RV-LTL monitor.

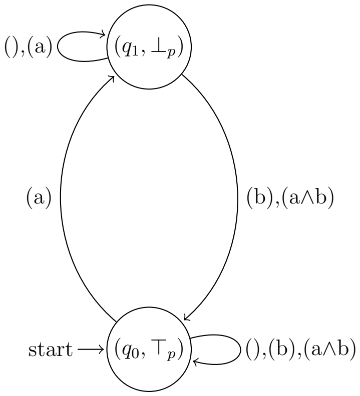


Figure 7.1: RV-LTL Monitor for $\varphi = \Box(a \rightarrow \Diamond b)$

The second half of the algorithm, starting at Line 18, processes the list generated at Line 14. If at least one combination of a loss-tolerant cluster and corresponding loss-tolerant alphabet satisfies the criteria of Definition 10 is found at Line 11, then φ may be monitored in presence of a transient loss of a sequence of events. In such case, the function AddUnknownStatetoCluster is called at Line 19 to synthesize a loss-tolerant monitor by augmenting $\mathcal{M}^{\varphi}$ with new states and transitions. For each combination, if a transition to one of the states in the corresponding loss-tolerant cluster is not defined for input χ, then a new state is created with output as '?'. The states in the loss-tolerant cluster then perform a transition to the newly added state, when the input is χ. Further, for each input element in the corresponding loss-tolerant alphabet, the new state makes transitions to one of the states in the loss-tolerant cluster. Also, for each of the input element, which does not belong to the loss-tolerant alphabet, the new state performs a transition to itself. FindMonitorablity connects all other non-final states that are not in the list of loss-tolerant clusters, identified at Line 14, to a common unknown state as seen at Line 21.

To further understand Algorithm 1, let us consider the example Ltl formula $\varphi = \Box(a \rightarrow \Diamond b)$ and its two-state monitor, i.e., $Q = \{q_0, q_1\}$, shown in Figure 7.1. The procedure FindMonitorablity identifies the tuple $(\{q_0, q_1, \}, \{a \wedge b, a, b\})$ as the only combination of a loss-tolerant cluster and the corresponding loss-tolerant alphabet that meets the required criteria in Definition 10. The procedure AddUnknownStatetoCluster adds a new state labeled as 'unknown' to the identified loss-tolerant cluster. This new state has the output symbol as '?' denoting that its verdict is unknown. For every state in cluster $\{q_0, q_1\}$, a new transition is added for the input symbol χ, and such transitions yield the new state. The transitions of the new state for all input symbols in Σ_{-} are same

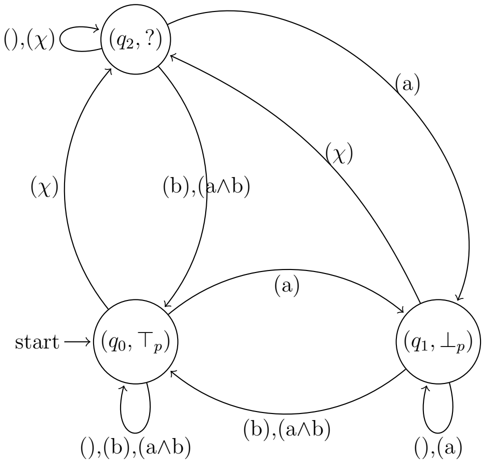


Figure 7.2: Loss-tolerant Monitor for $\varphi = \Box(a \rightarrow \Diamond b)$

as that of any of the states in the cluster. For every input element, which does not belong to Σ_{-} but belongs to Σ, a new transition is added for the new state such that it yields the new state itself. The loss-tolerant RV-LTL monitor for φ is as shown in Figure 7.2.

As discussed by Bauer et al. [13], the computation time of the procedure SynthesizeRvltlMonitor is exponential w.r.t. the size of the Ltl formula. If we consider the size of the alphabet Σ to be m and the number of states in a RV-LTL monitor $\mathcal{M}^{\varphi}$ to be n, then the complexity of the function MeetsMonitorability is $O(m \times n)$. Similarly, the complexity of AddUnknownStatetoCluster is $O(m + n)$. Thus, the worst-case time complexity of the inner section of Algorithm 1, i.e, from Line 8 to Line 17, is $O(m \times n \times 2^n)$. Therefore, the time complexity of Algorithm 1 for synthesizing a loss-tolerant monitor is exponential w.r.t. the number of states of the RV-LTL monitor. Since the loss-tolerant monitor is synthesized offline, this complexity is not as critical as the runtime complexity.

For RV, the additional overhead in terms of memory and CPU consumption at run time needs to be reduced. The number of states in the monitor is an important parameter as the states and their transitions consume memory at run time. The number of new states that can be added to the RV-LTL monitor is always bounded by n. Therefore, the size of the synthesized loss-tolerant monitor $\mathcal{M}^{\varphi}$ is also in $O(2^{2^p})$, where p is the size of the formula φ. Thus, the complexity for the size of loss-tolerant monitor is identical to that of the RV-LTL monitor [13].

Algorithm 1 Part 1 of Procedure to Verify Monitorability and Synthesize a Loss-tolerant Monitor for an LTL formula, φ

```
Input: $\varphi$                                              ▷ $\varphi$ is an LTL formula
Output: $(isMonitorable, \mathcal{M}^\varphi)$ ▷ $isMonitorable$ indicates whether $\varphi$ is monitorable, If $\varphi$
    is monitorable, then $\mathcal{M}^\varphi$ is loss-tolerant monitor. Otherwise, it's a RV-LTL Monitor
procedure FindMonitorablity($\varphi$)
                 ▷ Find whether $\varphi$ is monitorable in presence of a transient loss of events
    $isMonitorable \leftarrow \bot$
    $\mathcal{M}^\varphi \leftarrow$ SynthesizeRvltlMonitor($\varphi$)            ▷ Procedure in [13]
    $n \leftarrow$ FindNoOfStates($\mathcal{M}^\varphi$)
    if $n \leq 1$ then
        return $(isMonitorable, \mathcal{M}^\varphi)$
    end if
    for $i \leftarrow 2, n$ do
        for $j \leftarrow \binom{n}{i}, 1$ do
            $Q_s \leftarrow$ next-cluster of size $i$ of states of $\mathcal{M}^\varphi$
            $(ms, \Sigma_-) \leftarrow$ MeetsMonitorability($Q_s$)
            if $ms \neq \bot$ then
                $isMonitorable \leftarrow \top$
                $clusterList \leftarrow$ AddClusterToList($Q_s, \Sigma_-$)
            end if
        end for
    end for
    for each $(Q_s, \Sigma_-) \in clusterList$ do
        AddUnknownStatetoCluster($Q_s, \Sigma_-$)
    end for
    Create a new $chiState$ such that $\lambda(chiState) =$ '?'
    for $q \mid \forall Q_s \in clusterList \cdot getClusters() \cdot q \notin Q_s$ do
        if $q$ is not a final state then
            $\delta(q, \chi) = chiState$
        end if
    end for
    for each $\beta \in \Sigma \cup \chi$ do
        $\delta(chiState, \beta) = chiState$
    end for
    return $(isMonitorable, \mathcal{M}^\varphi)$
end procedure
```

Algorithm 2 Part 2 of Procedure to Verify Monitorability and Synthesize a Loss-tolerant Monitor for an Ltl formula, φ

```
procedure AddUnknownStatetoCluster($Q_s, \Sigma_-$)    ▷ Add ‘unknown’ state to a loss-tolerant cluster
    Create a new $chiState$ such that $\lambda(chiState) =$ ‘?′
    Set $q =$ any state $s$ such that $s \in Q_s$
    for each $\beta \in \Sigma_-$ do
        Set $\delta(chiState, \beta) = \delta(q, \beta)$
    end for
    for each $q \in Q_s \cup chiState$ do
        if $\delta(q, \chi)$ undefined then
            Set $\delta(q, \chi) = chiState$
        end if
    end for
    for each $\beta \notin \Sigma_- \wedge \beta \in \Sigma \cup \chi$ do
        Set $\delta(chiState, \beta) = chiState$
    end for
end procedure


procedure MeetsMonitorability($Q_s$)
    if ($Q_s$ satisfies Definition 10) then    ▷ This check is performed using elements of $\Sigma \setminus \chi$ because $\chi$ is not a part of the alphabet for the RV-LTL monitor
        Find $\Sigma_-$ for $Q_s$
        return $(\top, \Sigma_-)$
    else
        return $(\bot, null)$
    end if
end procedure
```

7.1.1 Correctness of Loss-tolerant Monitors

A loss-tolerant RV-LTL monitor $\mathcal{M}^{\varphi}$ for an Ltl formula φ operates on a trace σ, which is identical to the trace processed by the original monitor, with the exception that all the input elements in a bounded interval are unknown. We denote the value of such elements by χ. We also refer to such bounded interval as lossy interval. As stated by Lemma 1, the proof shows that when a loss-tolerant RV-LTL monitor processes an execution trace, which is same as that of the trace processed by RV-LTL monitor except for the lossy interval, the verdict produced by the loss-tolerant RV-LTL monitor is equal to that of the RV-LTL monitor provided that certain conditions are satisfied. These conditions are described in the following lemma.

Lemma 1 (Conditional Equality of Verdicts)**.** *Let* $M^{\varphi} = (\Sigma \setminus \chi, Q, q_0, \delta, \lambda)$ *be an RV-LTL monitor for the formula* φ *to process a trace* $\sigma_i = a_1, a_2, \ldots, a_n$ *and* $M'^{\varphi} = (\Sigma, Q', q'_0, \delta', \lambda')$ *be the corresponding loss-tolerant RV-LTL monitor to process a trace* $\sigma_j = b_1, b_2, \ldots, b_n$*.* $\exists!\, I = [i_1, j_1]$ *s.t.* $\forall k : (a_k = b_k) \vee (i_1 \leq k \leq j_1 \wedge b_k = \chi) \wedge (j_1 < n)$*. The verdict of* M'^{φ} *is conditionally equal to* M^{φ} *iff:* $\exists (Q_s, \Sigma_-) \cdot \exists m \in (j_1,\, n] \cdot \forall k \in [1, n]\ (((b_{i_1} = \chi \wedge q'_{i_1 - 1} \in Q_s) \wedge (b_m \in \Sigma_-)) \iff ((k \geq m \vee k < i_1) \wedge \lambda(\delta(q_{k-1}, a_k)) = \lambda'(\delta'(q'_{k-1}, b_k))) \vee (i_1 \leq k < m \wedge \lambda'(\delta'(q'_{k-1}, b_k)) = \text{‘}?\text{'})))$*, where* (Q_s, Σ_-) *is one of the combinations of a loss-tolerant cluster and the corresponding loss-tolerant alphabet identified by Algorithm 1.*

Proof. We prove the first part of Lemma 1. The first part is stated as $\exists (Q_s, \Sigma_-) \cdot \exists m \in (j_1,\, n] \cdot \forall k \in [1, n]\, (((b_{i_1} = \chi \wedge q'_{i_1 - 1} \in Q_s) \wedge (b_m \in \Sigma_-)) \longrightarrow ((k \geq m \vee k < i_1) \wedge \lambda(\delta(q_{k-1}, a_k)) = \lambda'(\delta'(q'_{k-1}, b_k))) \vee (i_1 \leq k < m \wedge \lambda'(\delta'(q'_{k-1}, b_k)) = \text{‘}?\text{'}))$

Firstly, let us *assume* that the input element being processed by M'^{φ} at the beginning of the lossy interval is χ, and M'^{φ}'s corresponding state belongs to one of the loss-tolerant clusters identified by Algorithm 1, and one of the input elements, b_m, processed by the monitor after the lossy interval belongs to the corresponding loss-tolerant alphabet.

If the current input element is χ and M'^{φ}'s state at the beginning of the lossy interval belongs to a loss-tolerant cluster added by Algorithm 1, then the next state of M'^{φ} is one of the new state with output as ‘?’. This new state is earlier added to the loss-tolerant RV-LTL monitor by Algorithm 1 during construction of the loss-tolerant monitor by procedure AddUnknownStatetoCluster. The *post-conditions* of this procedure ensure that for all states in the corresponding loss-tolerant cluster, an input χ yield a transition to a state with output as ‘?’. Further, M'^{φ} remains in this state until it can process an element, $b_m \in \Sigma_-$. Hence, its output remains ‘?’ when $i_1 \leq k < m$.

Further, when M'^{φ} observes b_m after the end of the lossy interval, it performs a transition to a state, which is equal as that of M^{φ} corresponding state as per the transition function defined by procedure AddUnknownStatetoCluster at Line 35. The *post-conditions* of procedures MeetsMonitorability and AddUnknownStatetoCluster ensure that such equality between the states of M^{φ} and M'^{φ} exists. Thus, for $k \geq m$, M'^{φ}'s output remains equal to that of M^{φ}. Further, when $k < i_1$, i.e., before the beginning of the lossy interval, the output of M'^{φ} is same as that of M^{φ} as both perform identical transitions with identical input symbols. Hence, the first part of the lemma is proved by our findings in this and previous paragraph.

We also prove the second part of Lemma 1. The second part is stated as $\exists(Q_s, \Sigma_-) \cdot \exists m \in (j_1, n] \cdot \forall k \in [1, n] ((((k \geq m \vee k < i_1) \wedge \lambda(\delta(q_{k-1}, a_k)) = \lambda'(\delta'(q'_{k-1}, b_k))) \vee (i_1 \leq k < m \wedge \lambda'(\delta'(q'_{k-1}, b_k)) = \text{`}?\text{'})) \implies ((b_{i_1} = \chi \wedge q'_{i_1-1} \in Q_s) \wedge (b_m \in \Sigma_-)))$

So, let us assume the output symbol of M'^{φ} is equal to that of M^{φ}, when $k < i_1 \vee k \geq m$ and M'^{φ}'s output is '?', when $i_1 \leq k < m$.

As the output of M'^{φ} is '?' for $i_1 \leq k < m$ and otherwise equal to that of M^{φ}, it is evident that $b_{i-1} = \chi$ as per the construction of monitor by Algorithm 1 where an unknown state has its output function as '?'. So, the lossy interval begins at $i-1$ because the output of M'^{φ} is '?' starting at index i_1. Further, as the output of M'^{φ} and M^{φ} is equal for m^{th} element of the trace, b_m must belong to one of the loss-tolerant alphabet, and the state of M'^{φ} before processing ${i_1}^{th}$ element must belong to the corresponding loss-tolerant cluster. Unless the state of M'^{φ} before processing ${i_1}^{th}$ element belongs to the loss-tolerant cluster, M'^{φ} cannot produce an output which is equal to that of M^{φ} upon processing the input element b_m, which belongs to the loss-tolerant alphabet. Thus, the second part of the lemma is proved as well. □

Further, we provide proof of correctness for loss-tolerant RV-LTL monitors constructed using Algorithm 1.

Theorem 1 (Correctness of M'^{φ})**.** *Let φ be an* Ltl *formula, and M'^{φ} be the loss-tolerant RV-LTL monitor constructed using Algorithm 1. For all $\sigma \in \Sigma^*$, if σ exhibits a transient loss of events as per Definition 7, and a run of M'^{φ} on σ satisfies (1) the state of the monitor immediately before the beginning of a lossy interval belongs to one of the loss-tolerant clusters and (2) the subsequent trace after the end of the lossy-interval contains at least one element from the corresponding loss-tolerant alphabet, then the following holds*

$$\lambda(\delta(q'_0, \sigma)) = [\sigma \models \varphi]_{RV}$$

Proof. Bauer et al. have provided the proof of correctness for LTL_3 [14] and RV-LTL [13] monitors. Our proof directly follows from Lemma 1, Definition 7 and the proofs provided by Bauer et al. □

In case of runtime verification of an incomplete trace, it is important that the verdict of the monitor cannot be invalidated upon the recovery of the lost events in the trace. To express this idea, we define *monotonicity* of loss-tolerant monitors w.r.t. events lost in the past. We note that the we only consider the events that are lost in the past. Thus, the verdict of a monitor may change when events are appended to a finite trace. Indeed, *monotonicity* allows that the verdict of the monitor does not depend upon the recovery of lost events. In general, a recovery of lost events may not be possible for real-world applications.

Definition 11 (Monotonicity w.r.t. past events)**.** *A monitor $\mathcal{M}^{\varphi}$ is monotonic iff its verdict on a finite word exhibiting loss cannot be invalidated by the recovery of the lost events in the finite word.*

Corollary 1. *A loss-tolerant monitor constructed using Algorithm 1 is monotonic.*

Proof. This proof trivially follows from that of Theorem 1 and Lemma 1. □

7.1.2 Performance

We measured the performance of Algorithm 1 on the commonly used patterns of LTL formulas identified in [14, 29]. A total of 42 formulas were identified as monitorable from the 97 LTL formulas. Figure 7.3 compares the number of states for the RV-LTL monitors and corresponding loss-tolerant monitors. It shows that the additional overhead incurred by the loss-tolerant monitor is not significant.For a loss-tolerant monitor, this number of states represents only an increase of at most two from that of the corresponding RV-LTL monitor. Similarly, Figure 7.4 compares the number of transitions between the monitors. The number of transitions depends upon the number of states and the size of the alphabet. The number of transitions depicts the additional overhead in terms of memory at run time. We observe a minimum of 5 and a maximum of 534 extra transitions w.r.t the number of transitions in the RV-LTL monitors.

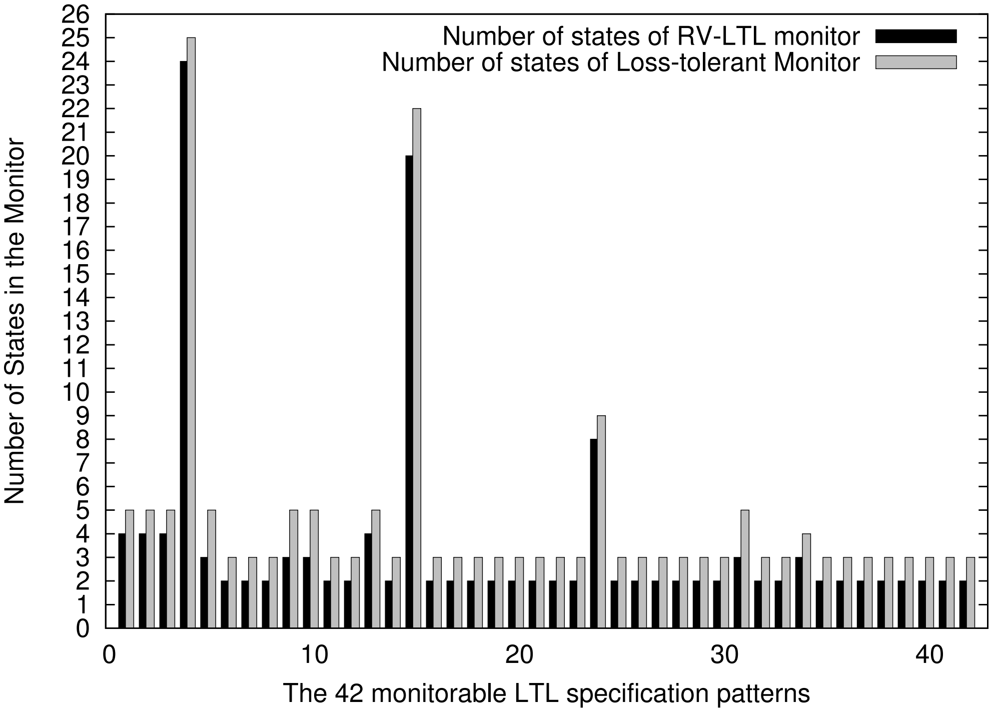


Figure 7.3: Comparison of the No. of States

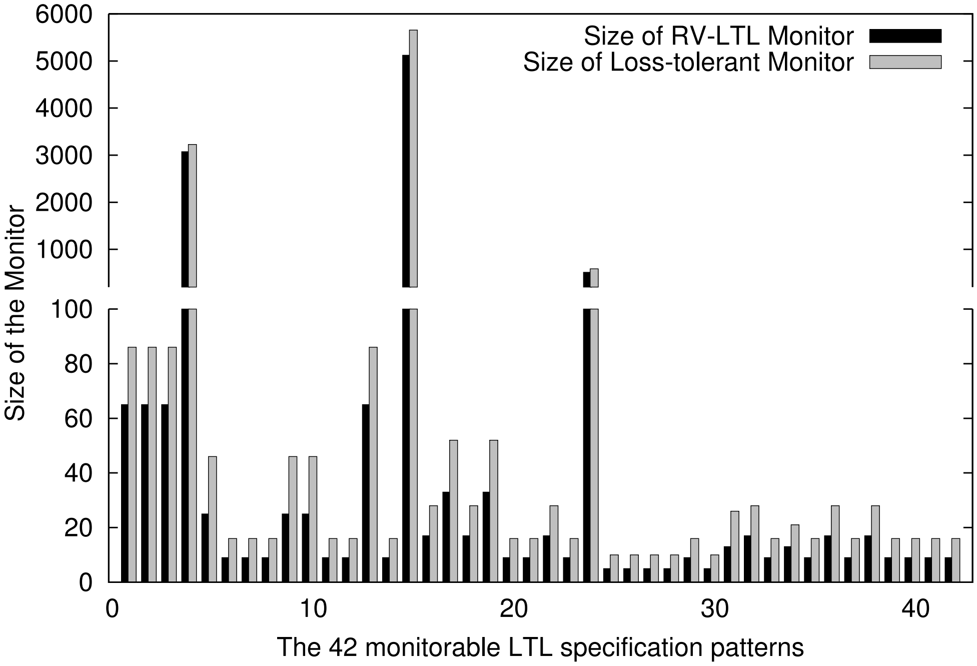


Figure 7.4: Comparison of the No. of Transitions

Chapter 8

Case-studies for Loss-tolerant Monitoring

In this chapter, we present two case studies depicting various results about our appproach. We evaluated our approach for monitoring MPlayer version 1.1_4.8 [62], which is a cross-platform media player application, and Google cluster-usage traces [69, 78]. For the experiments, we used a single eight core machine equipped with the Intel i7-3820 CPU at 3.60GHz and 31.4Gb of RAM. The machine runs Ubuntu 14.04 LTS 64 bit. We implemented our loss-tolerant monitors in Java using RV-LTL monitors generated using LamaConv [71].

8.1 MPlayer

Our goal here is to perform online monitoring while MPlayer plays a high definition, 29.97 fps, 720x480, 1 Mbps bitrate video. We evaluate the following LTL properties.
Property 1: $\varphi_1 :: \Box \mathit{play} \rightarrow \Box(\mathit{buffer} \rightarrow \Diamond \mathit{decode})$. This property saids that whenever MPlayer plays a file, a buffer action is always followed by an eventual decode action.
Property 2: $\varphi_2 :: \Box(\mathit{play} \rightarrow \Diamond(\mathit{audio} \vee (\mathit{video} \wedge \mathit{subtitle}) \vee \mathit{pause}))$. this expresses that when MPlayer plays a file, it may be an audio file or a video with subtitle or just paused.

8.1.1 Settings

We use DIME [3], which is a flexible time-aware dynamic binary instrumention tool as the observer. DIME uses a rate-based resource allocation method to periodically extract information from a program while still preserving its timing constraints. To achieve this, DIME limits the instrumentation time to a predefined budget B during each period of T time units. The instrumentation is then performed for no more than B time units per period T. The instrumentation budget is reset to the full at the beginning of each new period. When the budget is fully consumed, the instrumentation is disabled and re-enabled again only at the next period when the reset happens. Thus, it might be impossible for DIME to instrument at all code locations, causing incomplete logging or loss of events. DIME is implemented on top of Pin [54], a cross-platform dynamic binary instrumentation framework. Unlike DIME, Pin always instruments all code locations, providing a trace without any loss of events. Naturally, we used Pin to generate a complete trace.

We instrumented MPlayer with both DIME and Pin to respectively generate lossy traces and complete traces. DIME is extended to output a χ character when instrumentation is disabled. DIME can compress a set of consecutive χ characters into a single character. This compression also helps to counter the overhead of runtime verification. DIME collects and sends data to the monitor using a named pipe for online monitoring.

8.1.2 Results

We instrumented MPlayer with both DIME and Pin to track function calls related to Property 1 and Property 2. We varied the budget of DIME by 10 from 10% to 100% and repeated each experiments five times. The instrumentation period is fixed to 1 second. The monitors can buffer up to 100 events before processing them. Figure 8.1 shows that the output of the loss-tolerant monitor matches that of the RV-LTL monitor except the χ symbol. This figure plots a portion of the truth-values generated by the monitors for the trace obtained from the first run of DIME with 10% budget and Pin. The x-axis represents the index of each symbol in the sequence of input events, the y-axis, the truth values. For example, the verdict of the synthesized monitor for the 55^{th} input symbol is '?' while that of the RV-LTL monitor is $\top_P$, suggesting that the input was a χ. Figure 8.1 also shows that compression is disabled on the loss symbol to demonstrate the soundness of our monitoring algorithm. The final verdicts of the loss-tolerant monitor and that of the RV-LTL monitor are $\top_p$ for the described properties.

Further, we analyzed the monitoring overhead of each monitor. We turned on compression on the synthesized RV-LTL that treats a sequence of χ symbols as a single input and consequently produces only a single output. Figure 8.2 compares the average execution times of the two monitors for each property. The instrumentation budget appears on the x-axis and the total monitoring time on the y-axis. Figure 8.2a and Figure 8.2b both agree

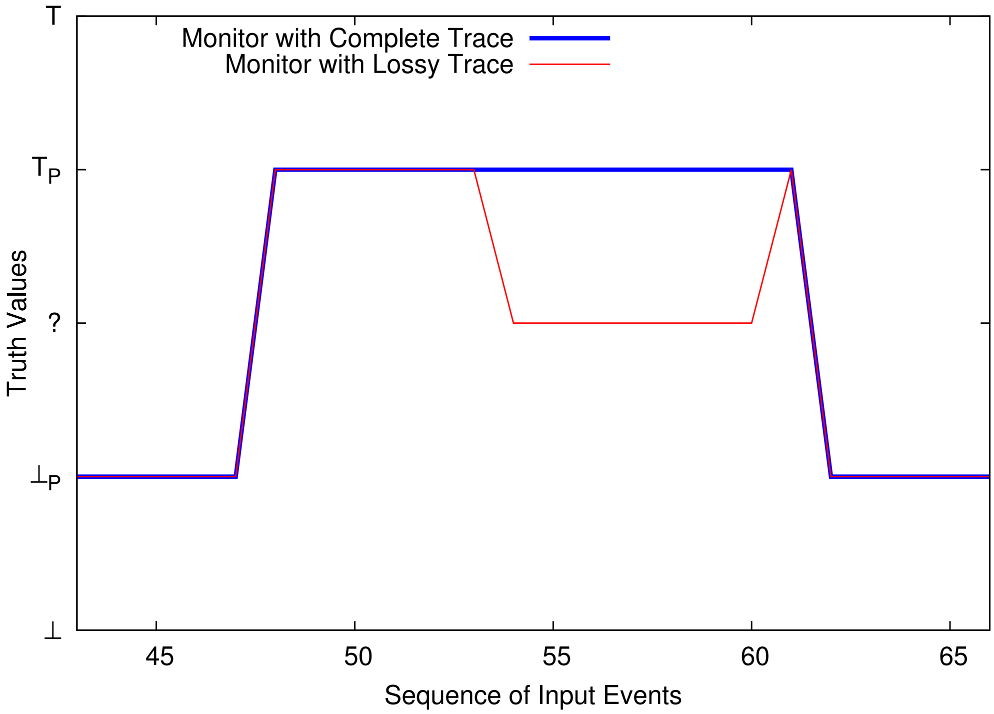
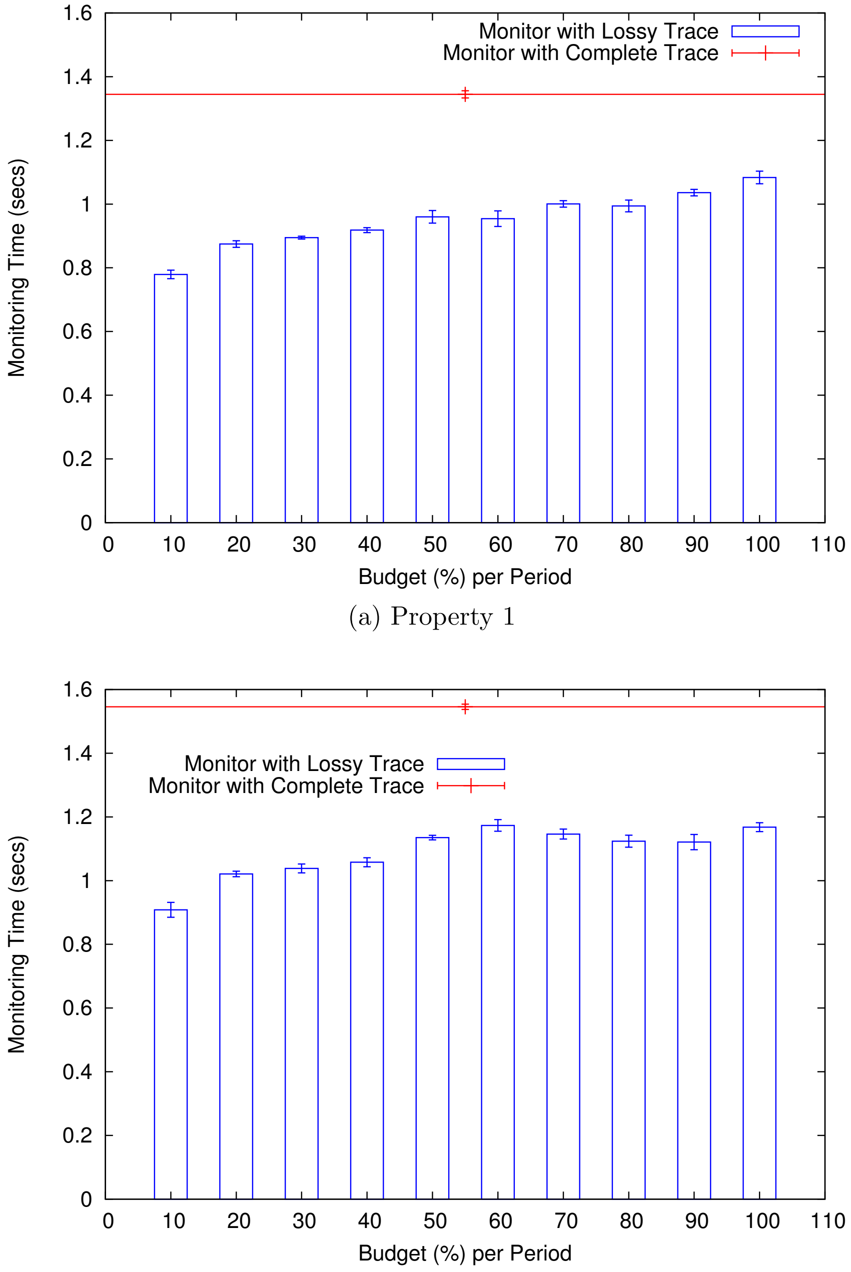


Figure 8.1: Comparison of Verdicts



(a) Property 1

(b) Property 2

Figure 8.2: Monitoring Overhead

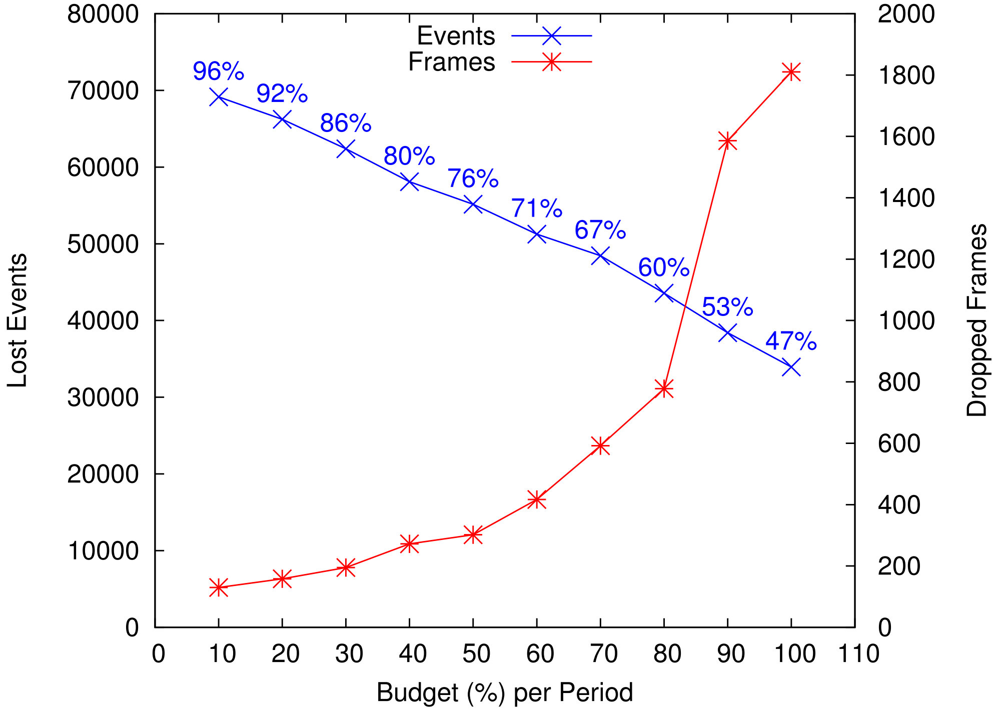


Figure 8.3: Dropped Frames vs Lost Events

on the fact the it is always more expensive to monitor a complete trace than to monitor an execution trace with transient loss. We can observe that the total monitoring overhead slightly increases with the instrumentation budget. Figure 8.2 shows that Property 1 and Property 2 can be soundly monitored even with a minimal instrumentation of 10% per second. Thus, for loss-tolerant LTL properties, one can decide to trade-off instrumentation overhead and throughput by skipping some events with the guarantee that the properties will be soundly monitored still.

To verify this assertion, we compare the number of lost events to the number of video frames dropped by the MPlayer. Figure 8.3 shows the results for Property 1. The x-axis represents the instrumentation budget per period of 1 second. The total number of lost events appears of the left y-axis, while the right y-axis represents the number of dropped frames. The numbers shown in Figure 8.3 represents the percentage of lost events w.r.t. the total number of processed events. We observe that the number of dropped frames increases with the instrumentation budget, while on the contrary, the number of lost events decreases as the instrumentation budget increases. With the budget at 80% for example, the number of lost events is around 60% with 1000 frames dropped. When the budget is at 100%, DIME still generate about 47% of loss in the execution trace. This is due to the rate-based budget allocation policy that forces DIME to always disable instrumentation once the budget is consumed, and to re-enable it only when the budget is replenished. As we increase the budget, more frames are dropped and the video becomes more and more sluggish as with Pin that may drop up to 3000 frames to generate a complete trace. Figure 8.3 suggests that for a smooth video experience, it is preferable to monitor while instrumenting with a budget of 10% per second.

8.2 Google Cluster-usage Traces

A Google cluster is a set of machines, packed into racks, and connected by a high-bandwidth cluster network [69, 78]. Work is performed on the cluster in the form of jobs, each comprising one or more tasks. Tasks and jobs are scheduled onto different machines. The ClusterData2011_2 [69, 78] used in this thesis, provides data from an 12.5k-machine cell over about a month-long period in May 2011. Data in the trace is derived from monitoring data, collected by periodic remote procedure calls. Therefore, when the monitoring system or cluster gets overloaded, data may not be collected. However, when an event record is missing in the trace, a replacement is synthesized with a note marking the details as 'missing information' [69, 78].

Each task progresses through different states such as 'submitted', 'scheduled', 'failed', 'evicted' and 'killed'. This data is lossy because some of such states for some jobs and their tasks are not captured due to unknown reasons. We extracted $7,170,572$ events for $16,467$ tasks for which some information about certain events is missing or incomplete. Following properties were monitored for verification of the state transitions of these tasks.
Property 1: $\Box((\mathit{fail} \vee \mathit{kill} \vee \mathit{evict}) \rightarrow \Box(\mathit{submit} \rightarrow \Diamond \mathit{finish}))$. This states that whenever a task fails or is killed or is evicted, it is always re-submitted, and eventually finishes.
Property 2: $\Box(\mathit{evict} \rightarrow \Diamond \mathit{submit})$. Means that whenever a task is evicted, it is eventually re-submitted.
Property 3: $\Box(\mathit{schedule} \rightarrow \Diamond(\mathit{finish} \vee \mathit{fail} \vee \mathit{kill} \vee \mathit{evict}))$. This states that whenever a task is scheduled, it will eventually finish, fail, get killed or evicted.
Property 4: $\Box(\mathit{update\_pending} \rightarrow \Diamond \mathit{schedule})$. This states that whenever a task is updated at runtime before been scheduled, it is eventually scheduled for execution.
Property 5: $\Box(\mathit{submit} \rightarrow \Diamond \mathit{finish})$. This states that whenever a task is submitted, it eventually gets finished.

The monitoring results for the above properties appears in Table 8.1. We report the

Prop.	**Monitored Tasks**	**Skipped Tasks**	**Verdict**	
			$\top_p$	$\bot_p$
1	10,847	5,620	3,572	7,275
2	10,892	5,575	10,891	1
3	10,892	5,575	6,240	4,652
4	16,467	0	16,461	6
5	10,892	5,575	2,876	8,016

Table 8.1: Verification of Google Cluster Data

number of tasks that were monitored. Some of the tasks were not monitorable because the respective traces did not match the criteria for monitorability. This criteria varies for the respective properties. For Property 1 for example, the loss-tolerant monitor delivered verdict as $\top_p$ for $5,620$ tasks and $\bot_p$ for $3,572$ out of the $10,847$ of tasks processed. A total of $5,620$ tasks were skipped because the traces did not satisfy the monitorability criteria for the property. For Property 2, the results show that most of the tasks, which are evicted, get resubmitted.

These results reveal that many of the tasks with incomplete traces do not successfully finish. Figure 8.4 shows that the average overhead caused by the loss-tolerant monitors for respective properties is not significant.

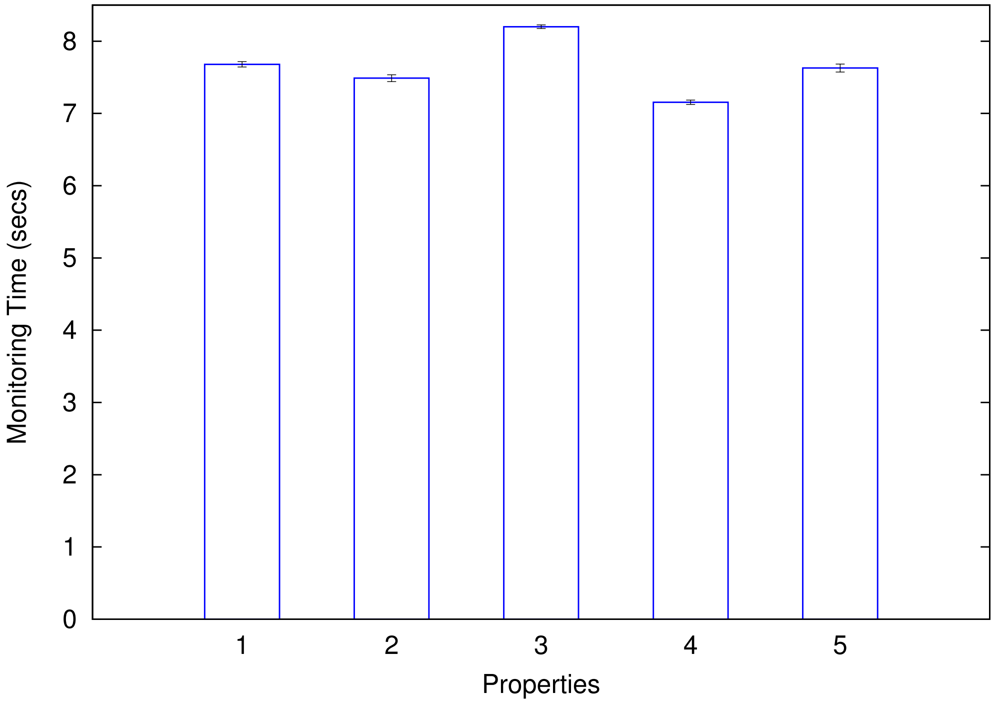


Figure 8.4: Monitoring Overhead of Google Traces

Chapter 9

Future Work and Conclusion

9.1 Future Work

In future, we plan to extend RitHM by adding plugins for executing monitoring algorithms on FPGAs. FPGAs offer a way of implementing parallel algorithms. We plan to develop monitor generators, which synthesize monitoring code in different programming languages. Further, we intend to come up with a web repository of different types of specifications with efficient mining and matching functions in order to develop a database of commonly used specification patterns. We believe that such repository can assist in the development of specification mining algorithms, which primarily rely on traces.

We also plan to extend RitHM with additional overhead control schemes based on power-efficient algorithms for monitor invocation. Further, we would like to develop additional RitHM plugins for monitoring using JMX, which is a widely used technology for monitoring large-scale enterprise Java applications. We plan to integrate monitoring algorithms for verification of distributed systems, which exhibit several additional problems related to out-of-order delivery of messages and lossy channels for inter-process communication.

For the problem of monitoring lossy traces, we plan to evaluate our monitoring method for distributed systems that may often produce incomplete and lossy traces. We also plan to extend our approach for mining specifications using incomplete traces as well as for monitoring variants of first-order temporal logic. Additionally, we plan to adapt our approach for different models of lossy traces such as lossy traces with a bound on the maximum number of lost events. We plan to evaluate our approach for such models, which depict different real-world applications which produce lossy traces under different constraints on the loss.

9.2 Conclusion

We presented RitHM, a comprehensive framework for scalable and efficient runtime verification of software. RitHM's design is motivated by state-of-the-art research on runtime verification, and it is extensible due to RitHM's support for plugins. RitHM exhibits several design features in order to support efficient monitoring by using different back-ends and parallel verification algorithms. Additionally, RitHM supports plugins for front-ends in order to allow flexibility in choosing a suitable logical formalism for describing the correctness criteria. RitHM also supports interoperability between different implementations of monitoring algorithms, and this feature provides significant reduction in the resource utilization of monitors. Further, we evaluated RitHM's architecture and architectures of a few more tools using ATAM. The evaluation highlights the importance of various unique features of RitHM's architecture. Our empirical evaluation also confirms the importance of the aforementioned design features to achieve the goals namely *modifiability*, *efficiency*, *usability* and *portability*.

We also investigated the problem of runtime verification of LTL in presence of transient loss of events. We introduced the concept of monotonicity and identified a fragment of LTL that can be soundly monitored in the presence of such a loss. Based on this, we presented an offline algorithm which identifies whether a given LTL formula is monitorable in presence of a transient loss, and constructs the corresponding loss-tolerant monitor. We analyzed the complexity of the algorithm and also evaluated it against commonly used patterns of LTL formulas to show that the synthesized monitor incur minimal additional overhead in terms of number states and transitions. Further, we evaluated the algorithm on the traces of two real-world systems to show the effectiveness and applicability of our approach. The evaluation also suggests that for LTL formulas that are monitorable in presence of transient loss, the monitoring overhead and the throughput of the monitored program can be traded-off by sacrificing some events with the guarantee that the properties can still be soundly verified. The evaluation also highlights that our approach increases the applicability of runtime verification for real-world applications, which often produce incomplete traces.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to Aspectj. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364. ACM, 2005.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining Specifications. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002.

[3] Pansy Arafa, Hany Kashif, and Sebastian Fischmeister. DIME: time-aware Dynamic Binary Instrumentation using Rate-based Resource Allocation. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 25:1–25:10. IEEE, 2013.

[4] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *ACM Sigplan Notices*, volume 43, pages 143–162. ACM, 2008.

[5] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. Mapreduce for Parallel Trace Validation of LTL properties. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 184–198, 2012.

[6] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal*

Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[7] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zlinescu. Monitoring compliance policies over incomplete and disagreeing logs. In *Runtime Verification*, pages 151–167. Springer, 2013.

[8] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. MONPOLY: Monitoring Usage-Control Policies. In Khurshid and Sen [49], pages 360–364.

[9] David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. Monitoring of Temporal First-Order Properties with Aggregations. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2013.

[10] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring Metric First-Order Temporal Properties. *J. ACM*, 62(2):15, 2015.

[11] Andreas Bauer, Rajeev Goré, and Alwen Tiu. A First-Order Policy Language for History-Based Transaction Monitoring. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2009.

[12] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138. Springer, 2007.

[13] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[14] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[15] Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. GPU-based Runtime Verification. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 1025–1036. IEEE Computer Society, 2013.

[16] Eric Bodden. J-LO-A tool for runtime-checking temporal assertions. *Master's thesis, RWTH Aachen university*, 2005.

[17] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 43(1):29–60, 2013.

[18] Edward Chang, Zohar Manna, and Amir Pnueli. *Characterization of temporal property classes*, pages 474–486. Automata, Languages and Programming. Springer, 1992.

[19] R. N. Charette. Why Software Fails [Software Failure]. *IEEE Spectr.*, 42(9):42–49, September 2005.

[20] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[21] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[22] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009.

[23] Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[24] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174. IEEE Computer Society, 2005.

[25] Dipankar Dasgupta and Zbigniew Michalewicz. *Evolutionary Algorithms in Engineering Applications.* Springer, 1997.

[26] Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{\mathrm{RV}}$-Adding Runtime Verification to jUnit. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 459–464. Springer, 2013.

[27] Alexandre Donzé. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 167–170. Springer, 2010.

[28] M. Dwyer, G. Avnmin, and J. Corbett. A System of Specification Patterns. http://patterns.projects.cis.ksu.edu/, 1997.

[29] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999.

[30] G. Fairhurst and L. Wood. Advice to Link Designers on Link Automatic Repeat reQuest (ARQ), 2002.

[31] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, 2012.

[32] Long Fei and Samuel P. Midkiff. Artemis: Practical runtime monitoring of applications for execution anomalies. *ACM SIGPLAN Notices*, 41(6):84–95, 2006.

[33] Dov M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987.

[34] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):658–671, 2010.

[35] Hendra Gunadi and Alwen Tiu. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 296–311. Springer, 2014.

[36] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[37] Sylvain Hallé and Roger Villemaire. Runtime Verification for the Web - A Tutorial Introduction to Interface Contracts in Web Applications. In Howard Barringer, Yliès, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2010.

[38] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ACM Sigplan Notices*, volume 39, pages 156–164. ACM, 2004.

[39] Klaus Havelund and Grigore Rosu. Monitoring Programs Using Rewriting. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pages 135–143. IEEE Computer Society, 2001.

[40] Klaus Havelund and Grigore Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[41] Klaus Havelund and Grigore Rosu. Efficient Monitoring of Safety Properties. *STTT*, 6(2):158–173, 2004.

[42] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online Monitoring of Metric Temporal Logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014.

[43] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. ROSRV: Runtime Verification for Robots. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 247–254, 2014.

[44] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer*, 14(3):327–347, 2012.

[45] Yongkai Huo, C. Hellge, T. Wiegand, and L. Hanzo. A Tutorial and Review on Inter-layer FEC Coded Layered Video Streaming. *Communications Surveys Tutorials, IEEE*, 17(2):1166–1207, Secondquarter 2015.

[46] Java Management Extensions (JMX). https://docs.oracle.com/javase/tutorial/jmx/.

[47] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A Method for Analyzing the Properties of Software Architectures. In Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde, editors, *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, pages 81–90. IEEE Computer Society / ACM Press, 1994.

[48] Rick Kazman, Rick Kazman, Mark Klein, Mark Klein, Paul Clements, Paul Clements, Norton L. Compton, and Lt Col. Atam: Method for architecture evaluation, 2000.

[49] Sarfraz Khurshid and Koushik Sen, editors. *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*. Springer, 2012.

[50] Linghe Kong, Mingyuan Xia, Xiao-Yang Liu, Guangshuo Chen, Yu Gu, Min-You Wu, and Xue Liu. Data Loss and Reconstruction in Wireless Sensor Networks. *Parallel and Distributed Systems, IEEE Transactions on*, 25(11):2818–2828, Nov 2014.

[51] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[52] Martin Leucker. Teaching Runtime Verification. In Khurshid and Sen [49], pages 34–48.

[53] LTTng: An Open Source Tracing Framework for Linux. http://lttng.org/.

[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building

customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[55] Zohar Manna. *Temporal verification of reactive systems: safety*, volume 2. Springer Science & Business Media, 1995.

[56] Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Accelerated runtime verification of LTL specifications with counting semantics. *CoRR*, abs/1411.2239, 2014.

[57] Ramy Medhat, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. Sacrificing a Little Space Can Significantly Improve Monitoring of Time-sensitive Cyber-physical Systems. In *ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS, Berlin, Germany, April 14-17, 2014*, pages 115–126. IEEE Computer Society, 2014.

[58] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[59] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[60] Bongki Moon, Inés Fernando Vega López, and Vijaykumar Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. Knowl. Data Eng.*, 15(3):744–759, 2003.

[61] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. An asymptotically correct finite path semantics for ltl. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 304–319. Springer, 2012.

[62] MPlayer - The Movie Player. `http://www.mplayerhq.hu`, 2015.

[63] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. RiTHM: A tool for enabling time-triggered runtime verification for c programs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 603–606, New York, NY, USA, 2013. ACM.

[64] Rodolfo Pellizzoni, Patrick O'Neil Meredith, Marco Caccamo, and Grigore Rosu. Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*

2008, Barcelona, Spain, 30 November - 3 December 2008, pages 481–491. IEEE Computer Society, 2008.

[65] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[66] Shravan K. Rayanchu, Arunesh Mishra, Dheeraj Agrawal, Sharad Saha, and Suman Banerjee. Diagnosing Wireless Packet Losses in 802.11: Separating Collision from Weak Signal. In *Proceedings of the 27th Conference on Computer Communications*, INFOCOM'08, pages 735–743. IEEE, April 2008.

[67] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.

[68] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2012.03.20. http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[69] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google Cluster-usage Traces: Format + Schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2012.03.20. http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[70] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), 17-19 August 2005, Hong Kong, China*, pages 147–153, 2005.

[71] Torben Scheffel, Malte Schmitz, Sebastian Hungerecker, Marco Kabelitz, Christofer Krüger, and Johannes Thorn. LamaConv: Logics and Automata Converter Library. http://www.isp.uni-luebeck.de/lamaconv, 2015.

[72] Aamir Shafi, Bryan Carpenter, Mark Baker, and Aftab Hussain. A Comparative Study of Java and C Performance in Two Large-scale Parallel Applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009.

[73] K. Sollins. The TFTP Protocol (Revision 2). RFC 1350 (Standard), July 1992. Updated by RFCs 1782, 1783, 1784, 1785, 2347, 2348, 2349.

[74] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *Runtime Verification*, pages 193–207. Springer, 2012.

[75] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[76] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci.*, 113:145–162, 2005.

[77] Virginie Wiels, Rémi Delmas, David Doose, Pierre-Loïc Garoche, Jacques Cazin, and Guy Durrieu. Formal verification of critical aerospace software. *AerospaceLab Journal*, 2012.

[78] John Wilkes. More Google Cluster Data. Google research blog, November 2011. http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.