Runtime Monitoring for Uncertain Times

by

Sean Kauffman

A thesis
presented to the University Of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical & Computer Engineering

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Prof. Rajeev Alur
Supervisor	Prof. Sebastian Fischmeister
Internal Member	Assoc. Prof. Arie Gurfinkel
Internal Member	Asst. Prof. Mark Crowley
Internal-external Member	Prof. Joanne Atlee

Author's Declaration

Statement of Contributions

This thesis includes material from four published works ([130, 127, 125, 128]) completed under the supervision of Dr. Sebastian Fischmeister at the University of Waterloo, who contributed with ideas, discussion, editing and manuscript review.

Chapter 2 primarily consists of material published in [130], which I co-authored with Dr. Klaus Havelund and Dr. Rajeev Joshi. Most of the article was written collaboratively, but Section 2.5.2 was mostly written by Dr. Havelund and Section 2.6 was mostly written by Dr. Joshi. The work is a journal extension of an earlier conference paper [129] which realizes ideas originally proposed in [131]. An exception is Section 2.8, which was a separate publication [127] co-authored exclusively with Dr. Fischmeister. Section 2.5.4 has not been previously published.

Chapter 3 primarily consists of material published in [125], which I co-authored with Murray Dunne, Dr. Giovani Gracioli, Waleed Khan, and Nirmal Benann. Section 3.2.2 was originally conceived by Mr. Dunne and worked on by Dr. Gracioli and myself. Most of Section 3.3 was written collaboratively with Mr. Dunne and Dr. Gracioli, but Section 3.3.1 is based on work published in [77] by Mr. Dunne, Dr. Gracioli, and Dr. Fischmeister to which I did not contribute. The description of the vehicle at the beginning of Section 3.4 was written by Mr. Khan, and Mr. Benann wrote Section 3.4.2. Mr. Khan designed and executed the Siddhi part of the comparison in Section 3.4.3. Sections 3.5.1 and 3.5.2 were mostly written by Dr. Gracioli.

Chapter 4 primarily consists of material published in [128], which I co-authored with Dr. Klaus Havelund. Dr. Havelund contributed with ideas, editing, and critiques.

The research performed by Klaus Havelund and Rajeev Joshi and part of the research performed by Sean Kauffman was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by Klaus Havelund was furthermore supported by AFOSR Grant FA9550-14-1-0261.

Abstract

In Runtime Verification (RV), monitors check programs for correct operation at execution time. Also called Runtime Monitoring, RV offers advantages over other approaches to program verification. Efficient monitoring is possible for programs where static checking is cost-prohibitive. Runtime monitors may test for execution faults like hardware failure, as well as logical faults. Unlike simple log checking, monitors are typically constructed using formal languages and methods that precisely define expectations and guarantees. Despite the advantages of RV, however, adoption remains low.

Applying Runtime Monitoring techniques to real systems requires addressing practical concerns that have garnered little attention from researchers. System operators need monitors that provide immediate diagnostic information before and after failures, that are simple to operate over distributed systems, and that remain reliable when communication is not. These challenges are solvable, and solving them is a necessary step towards widespread RV deployment.

This thesis provides solutions to these and other barriers to practical Runtime Monitoring. We address the need for reporting diagnostic information from monitored programs with nfer, a language and system for event stream abstraction. Nfer supports the automatic extraction of the structure of real-time software and includes integrations with popular programming languages. We also provide for the operation of nfer and other monitoring tools over distributed systems with Palisade, a framework built for low-latency detection of embedded system anomalies. Finally, we supply a method to ensure program properties may be monitored despite unreliable communication channels. We classify monitorable properties over general unreliable conditions and define an algorithm for when more specific conditions are known.

Acknowledgements

This thesis would not have been possible without the support and guidance of many people. My supervisor, Sebastian Fischmeister, has opened many doors for me and I owe him an enormous debt for his confidence. My wife, Kaitlin Lindquist, has been tremendously patient with me and her unshakeable belief when I have doubted myself has sustained me. Finally, I could not have written this without Klaus Havelund, who has taught me a great deal about Computer Science, but who has also been a wonderful mentor and friend.

Dedication

This thesis is dedicated to my parents, Edward and Gay Kauffman, who have always encouraged my curiosity.

A Note on the Title

The title *Runtime Monitoring for Uncertain Times* is a play on the themes of this thesis and on its time of publishing. It refers to temporal relativity (inferring the relative timing of intervals), it refers to unreliable timing (monitoring over unreliable channels), and it also refers to the uncertain state of the world during the current global pandemic.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AA Acceptance Accuracy

ACC Adaptive Cruise Control

ADAS Advanced Driver-Assistance Systems

ADC Analog to Digital Converter

AFR Always Finitely Refutable

AFS Always Finitely Satisfiable

AI artificial intelligence

AIS Active Instance Stack

API Application Programming Interface

ATAM Architecture Tradeoff Analysis Method

ATL Allen's Temporal Logic

BIDE BI-Directional Extension

BJI between-job inter-arrival time

CAN Controller Area Network

CAPEC Common Attack Pattern Enumeration and Classification

CEP Complex Event Processing

CFSM communicating finite state machine

CPU	Central Processing Unit
CSV	Comma Separated Value
DFA	Deterministic Finite Automaton
DFT	Discrete Fourier Transform
DSL	domain-specific language
DSMS	Data Stream Management System
DUT	device-under-test
ECU	Electronic Control Unit
EDF	Earliest Deadline First
EGADS	Extendible and Generic Anomaly Detection System
EPL	Event Processing Language
ETM	Embedded Trace Macrocell
EVR	EVent Report
FA	Finite Automaton
FCDA	Forward Collision Detection and Avoidance
FIFO	first-in first-out
FR	Finitely Refutable
FS	Finitely Satisfiable
FSM	finite state machine
GCC	the GNU Compiler Collection
GIL	Graphical Interval Logic
GPLv3	Gnu Public License version 3
GPS	Global Positioning System

GSQL	Gigascope Query Language
GUI	Graphical User Interface
HDLC	High-level Data Link Control
HMM	Hidden Markov Model
HPC	High Performance Computing
IDS	Intrusion Detection System
IFP	Information Flow Processing
IJI	intra-job inter-arrival time
IMU	Inertial Measurement Unit
IP	Internet Protocol
ISR	interrupt service routine
ITL	Interval Temporal Logic
JIT	Just-In-Time
JPL	Jet Propulsion Laboratory
JSON	JavaScript Object Notation
k-NN	k-Nearest Neighbors
LANL	the Los Alamos National Laboratory
LANSCE	Los Alamos Neutron Science Center
LCFSM	lossy communicating finite state machine
LDW	Lane Departure Warning
LKA	Lane Keeping Assistance
LODA	Lightweight Online Detector of Anomalies
LTL	Linear Temporal Logic

LTL$_3$	Three-value Linear Temporal Logic (LTL)
LTL$_4$	four-value LTL
MCU	Micro-controller Unit
MSL	Mars Science Laboratory
MTL	Metric Temporal Logic
NASA	the National Aeronautics and Space Administration
NBA	Non-deterministic Büchi Automaton
NFA	Non-deterministic Finite Automaton
`NFR`	Never Finitely Refutable
`NFS`	Never Finitely Satisfiable
OS	Operating System
POCS	projections onto convex sets
PSL	Property Specification Language
PTL	Propositional Temporal Logic
PTP	Precision Time Protocol
QEA	Quantified Event Automata
RA	Rejection Accuracy
REMS	Rover Environmental Monitoring Station
RESP	the REdis Serialization Protocol
RM	Rate Monotonic
RNN	Recurrent Neural Network
ROS	Robot Operating System
RPN	Reverse Polish Notation

RV Runtime Verification

RV-LTL Runtime Verification LTL

SAAM Software Architecture Analysis Method

SAX Symbolic Aggregate approXimation

LANL System Call Logs with Natural Random Faults

SDK Software Development Kit

SDL Specification and Description Language

`SFR` Sometimes Finitely Refutable

`SFS` Sometimes Finitely Satisfiable

SoC System-on-a-Chip

SPACE Sequential PAttern Discovery using Equivalence classes

SPAM Sequential PAttern Mining

SQL Structured Query Language

SRV Stream Runtime Verification

SSPS Sequential Sense-Process-Send

stdev standard deviation

STL Signal Temporal Logic

TRE Timed Regular Expression

UDP User Datagram Protocol

μ**HML** μ-Hennessy-Milner Logic

UML Unified Modeling Language

WCET Worst-Case Execution Time

XML Extensible Markup Language

Chapter 1

Introduction

Runtime Verification (RV), also called Runtime Monitoring, has gained acceptance as a means to verify the correctness of software where static methods are impractical [149]. Meanwhile, the number of safety-critical systems too large to statically check is growing exponentially, as the compound annual growth rate of software these systems has been estimated at roughly 1.16 [111, 223, 113]. Even for systems where model checking is possible, RV carries benefits. In addition to design faults, Runtime Monitoring can detect execution faults caused by circumstances such as hardware failure or malicious attacks [153].

Figure 1.1 shows the components of a traditional RV system. The Monitored System (1) outputs an Execution Trace (2) made of symbols to the Runtime Monitor (3). The Runtime Monitor uses the Specification (4) to compute a Stream of Verdicts (5) that it outputs to an Operator or an Enforcement Mechanism (6). Monitored systems are usually embedded computers with real-time requirements. These systems are often safety-critical, meaning their failure may result in harm to persons. The execution trace may take several forms, including timed events representing state transitions, regularly sampled real numbers representing sensor readings, or propositional formulae representing state. Monitors are special programs running independently from the monitored system. In some cases, monitors may run on the same computer as monitored software, but monitors also may execute on a separate machine connected to the monitored system by a network. Specifications are often written in a formal language such as Linear Temporal Logic (LTL), but they may also be encoded directly in monitoring algorithms or learned from historical traces. Verdicts are usually drawn from a domain including Boolean values (*true* and *false*) but they may also come from a richer domain of calculated results. An Operator may use the verdicts to make decisions about the Monitored System, or an Enforcement Mechanism may override behaviors.

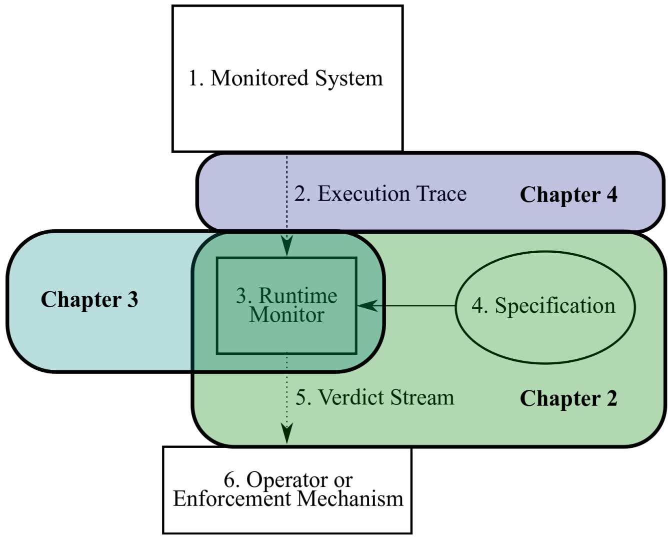


Figure 1.1: Components of a Runtime Verification system

Despite RV's potential, its adoption in many industries has been slow. In the automotive industry, where software can reach 20 million lines of code [46], even error logging for future diagnosis is uncommon, let alone comprehensive error checking or systematic error recovery [186]. Engineers in these and other industries have largely disregarded RV as a part of their safety strategy.

One reason so few have embraced RV is that research in the area has inadequately addressed many practical limitations. When deploying a Runtime Monitor in a realistic environment, some of these limitations become apparent. These problems affect every component of an RV system:

1. Most RV methods assume that the execution trace (2 in Figure 1.1) that reaches the Runtime Monitor (3) is the same trace output by the Monitored System (1). This assumption is often violated, however, when traces are corrupted by unreliable communication. Even when the monitor executes on the same computer as the monitored system, messages may be lost or modified.

2. For a Runtime Monitor (3 in Figure 1.1) to prevent failures, it must detect errors quickly enough for an operator or enforcement mechanism to react. However, monitoring algorithms for RV techniques often operate offline, meaning they require the system to have completed execution before they determine a verdict. Those monitors that operate online are often difficult to deploy and integrate into existing systems.

3. Specifications (4 in Figure 1.1) formalize the requirements of the system, but many systems do not have well-formalized requirements. RV research usually assumes that a specification is available and complete, but this is optimistic for all but the most rigorously defined projects. Cross-functional specifications are rarer still.

4. Most RV approaches result in a Verdict Stream (6 in Figure 1.1) containing only the status of whether a specification was violated or satisfied. If a violation occurs, the Operator or Enforcement Mechanism (6) may transition to a safe state, but no other information is provided to assist in error diagnosis or failure prevention.

This thesis proposes solutions to these and other practical barriers to the adoption of RV. These solutions are presented with the goal of increasing the use of formal verification for safety-critical software to help prevent harm caused by computing errors. Each chapter contains improvements to the state-of-the-art, motivated by the challenges faced when putting RV into practice.

In Chapter 2, we present `nfer`, a language and system for event stream comprehension. Unlike traditional verification tools, `nfer` is designed to produce an abstraction of program execution in the form of a hierarchy of temporal intervals. `Nfer` is a solution to Problem 4, that RV techniques often do not provide any information beyond success or failure. Other RV solutions exist that compute data on event streams, such as stream processors [155, 54, 103] and rule-based RV [22, 23], but `nfer`'s combination of computed facts with temporal intervals is unique. `Nfer` facilitates trace comprehension, providing crucial information to understand possible failures. `Nfer` is a rule-based system, but its rules may also be mined from real-time system traces. This is a solution to Problem 3, that RV methods usually require known, formalized system requirements. Users can discover new relationships in their data from these mined rules and use them to validate handwritten specifications. `Nfer` is also integrated with the popular programming languages R and Python, providing Application Programming Interfaces (APIs) that are easy to access without learning a domain-specific language (DSL).

Chapter 3 introduces Palisade, a framework for distributed online anomaly detection. Anomaly detection is a form of RV where patterns of nominal behavior are typically trained using historical data. These models of normal behavior are often created using statistical learning, but they may also be handwritten or created by other sources. Palisade is designed to operate multiple online anomaly detectors in parallel to address different potential symptoms of system error. Palisade is a solution to Problem 2, where RV monitors are often designed only to operate offline and do integrate easily with existing systems. Those RV systems that support online operation are highly specialized [216, 76], lack support for distributed operation [54, 103], or have high detection latencies [60, 217]. Palisade is both a communications and programming framework and solves many of the logistical hurdles to deploying RV in a realistic environment.

In Chapter 4, we identify what properties may be monitored in the presence of unreliable communication. This theoretical framework is a solution for Problem 1, where RV methods assume the integrity of execution traces. Prior work on unreliable channels in RV has focused on building monitoring systems that work under specific degraded conditions [232, 122, 27] or that quantify imprecision due to trace corruption [17, 214, 148]. Our solution is unique in that it is the only general framework for determining the monitorability of program properties. We model communication errors as relations of traces called mutations and propose general mutations for common circumstances. These mutations may cause monitors for some properties to provide incorrect verdicts, and we define what it means for a verdict to be trustworthy when a mutation may be present. We identify classes of properties with trustworthy verdicts for common mutations. We also provide an algorithm for testing if a property may be monitored in the presence of user-defined mutations.

Chapter 5 concludes the thesis and discusses future work. These contributions are a step towards solving practical limitations of RV that prevent its wider adoption. With continued efforts, we hope that RV can gain acceptance as a solution to verifying the correctness of safety-critical software.

1.1 Preliminary Notation

This section defines notation used throughout the thesis. Additional notation for Chapter 4 is defined in Section 4.2.

By $\mathbb{B}_2$ we denote the set of Boolean values $\{\mathit{true}, \mathit{false}\}$. For brevity, we sometimes write $\bot$ to denote *false* and $\top$ to denote *true*. We use $\mathbb{N}$ to denote the set of all natural numbers including zero $\{0, 1, 2, \ldots\}$ and ∞ to denote infinity. By $\mathbb{R}$ we denote the set of real numbers. For readability, we use the type $\mathcal{C}\mathit{lock} = \mathbb{R}$ to represent clock time stamps measured in continuous (or dense) time.

In this thesis, we consider both finite and infinite sequences. A finite sequence σ of n values is written $\sigma = \langle v_1, \cdots, v_n \rangle$ where both v_i and $\sigma(i)$ mean the i'th item in the sequence. Throughout the thesis, sequence index numbers begin at one. The notation $\langle v_1, v_2, \cdots \rangle$ is used to denote either an infinite sequence or a finite sequence of indeterminate length. A value x is in a sequence σ, denoted by $x \in \sigma$, iff $\exists\ i\ \in\ \mathbb{N}$ such that $\sigma(i) = x$. The length of a sequence σ is written $|\sigma| \in \mathbb{N} \cup \{\infty\}$. The suffix of a sequence σ beginning at the

i'th item in the sequence is written σ^i. A subsequence of σ beginning at and including the ith index and ending at and including the jth index is denoted $\sigma_{[i,j]}$. The concatenation of two sequences σ, τ is written $\sigma \cdot \tau$ where σ is finite and τ is either finite or infinite. A finite sequence u is a prefix of a finite or infinite sequence σ, written $u \sqsubseteq \sigma$, iff there exists a sequence v such that $u \cdot v = \sigma$.

By $A \times B$ we denote the cross product of sets A and B. By $A \rightarrow B$ we denote the set of total functions from A to B. Given a set S, we write 2^S to mean the set of all subsets of S. The cardinality of a set S is written $|S|$. Given a set S, S^* denotes the set of finite sequences over S where each sequence element is in S, S^ω denotes the set of infinite sequences of such elements, and $S^\infty = S^* \cup S^\omega$. Given a set S, by S^n for a given $n \in \mathbb{N}$ ($n \geq 2$) we denote the tuple type: $S \ \times \ S \ \times \ \ldots \ \times \ S$ (n times).

Let $\mathcal{I}$ be a set of names (identifiers, also called topics), and let $\mathcal{V}$ be a set of values, including strings, integers, and floating point numbers. A *map* is a partial function from names to values with a finite domain, that is, a function of type $\mathcal{I} \nrightarrow \mathcal{V}$. We use $\mathbb{M}$ to denote the type of all maps. The empty map is denoted by $[\,]$. We denote by $\mathbb{M}_\perp$ the extension of $\mathbb{M}$ with a bottom element: $\mathbb{M}_\perp = \mathbb{M} \ \cup \ \{\perp\}$. Here $\perp$ represents a "no map" value. Given $M \in \mathbb{M}$, $k \in \mathcal{I}$, and $v \in \mathcal{V}$, we write $M(k) \leftarrow v$ to denote the map M updated with k mapped to v.

Functions can be denoted by lambda terms: $\lambda x.e$. A function of type $A \rightarrow \mathbb{B}_2$ is referred to as a predicate. Predicates with the same domain type can be composed with Boolean operators. For example, given $f : A \rightarrow \mathbb{B}_2$ and $g : A \rightarrow \mathbb{B}_2$, then $(f \wedge g)(x) = f(x) \wedge g(x)$.

An event is a timestamped, named tuple of the type $\mathbb{E} = \mathcal{I} \ \times \ \mathcal{C}lock \ \times \ \mathbb{M}$. That is, it contains a name (sometimes called a topic), a clock time, and a map. An element (id, t, M) of type $\mathbb{E}$ may be written $id(t, M)$. In some cases, where maps are not needed, events may be written with maps omitted as (id, t). In this thesis, a trace may be a finite sequence of events (an *event series*), a finite sequence of symbols in an alphabet, or an infinite sequence of symbols in an alphabet. Where a trace means a finite event series, its type is denoted by $\mathbb{T}$ and is defined by $\mathbb{T} = \mathbb{E}^*$.

Figure 1.2 shows an example of a finite event series. In the figure, the event names are listed above the timeline and event times are listed below. Maps are omitted from the figure. Note that neither event names nor times must be unique in such a trace.

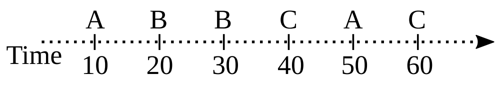


Figure 1.2: Example finite event series

Chapter 2

Abstracting Event Streams

2.1 Introduction

A key challenge in operating remote spacecraft is that human operators must rely on received telemetry to assess the status of the spacecraft. Telemetry can be thought of as an execution trace: a stream consisting of discrete events, each having a name, a time stamp, and carrying data. Telemetry streams can contain millions of events and can therefore be difficult to comprehend by humans, as well as interpret against the higher-level execution plans submitted to the spacecraft. At the National Aeronautics and Space Administration's (NASA's) Jet Propulsion Laboratory (JPL) the current approach to analyzing spacecraft telemetry for missions like Mars Science Laboratory (MSL), and specifically from its Curiosity rover, relies on ad-hoc scripts that are labor intensive to write and maintain. We propose a formalism for specifying *interval abstractions* of event streams, with a semantics that produces a set of intervals from a trace. Such abstractions can be useful for telemetry visualization[1] and querying to aid human comprehension. Our formalism is inspired by interval logics, specifically Allen's Temporal Logic [11], commonly used in the planning and artificial intelligence (AI) domains. We extend a variation of this logic with a rule-based declarative formalism, named `nfer`, for expressing event abstractions.

The `nfer` formalism and implementation has commonalities with classical rule-based systems known from AI. Our early work on the trace abstraction problem was effectively done using a rule-based system, as documented in [109, 108]. However, we learned that a

[1]Visualization of information is e.g. at JPL considered an important approach to aid humans in daily spacecraft operations.

rule system does not represent meta-level constraints well, i.e., properties that hold on the collection of all facts produced by the rule system. An example of a meta-level constraint is the notion of minimality: *do not create an interval if an interval has already been generated with the same name in the same time period.* Such meta-constraints and, in particular, the minimality constraint, turn out to be crucial to reduce the complexity of trace analysis. A similar observation can be made about Prolog. As an experiment, we formulated the chapter's guiding example in LogFire [107], our homegrown rule-based system used in [109, 108]; and in Prolog, and compared them to the Scala and C versions of `nfer`. The four systems were applied to an event trace of 10,000 randomly generated events. LogFire had to be aborted after running more than a week. Prolog took 64 hours to finish. The Scala version and C version, both of which implement minimality, finished in 1.8 and 0.05 seconds, respectively.

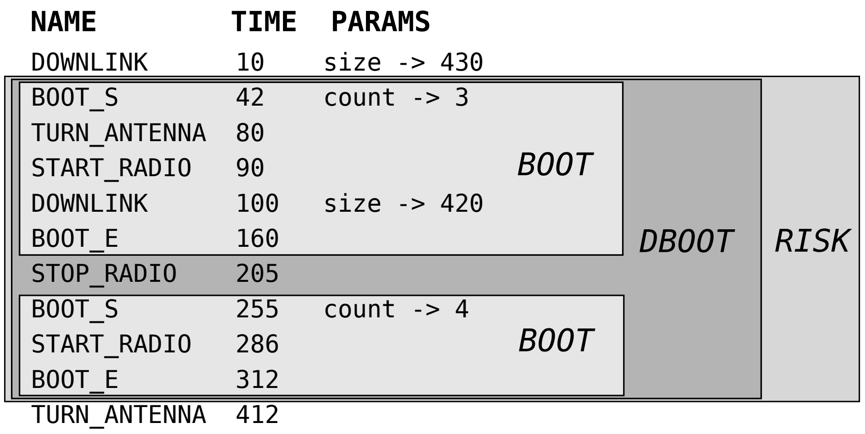
Our system differs from traditional runtime verification (RV) systems, in which a program execution trace is checked against a user-provided specification. RV usually results in a binary decision (true/false) as to whether the execution trace satisfies the specification, although variations on this theme have been developed, including 3-valued logics [47] and 4-valued logics [31]. In contrast, the result of running `nfer` on an event stream is a set of named and timed intervals carrying data collected from the trace, representing abstractions of the trace. Such abstractions can be visualized to support trace comprehension, or can be considered as input to further analysis.

This chapter primarily consists of work published in [130, 129] and [127]. We introduce the semantics and derived forms of `nfer` rules and propose a monitoring algorithm for them. We discuss two implementations of the monitoring system and study modifications to improve its scalability. We also present an algorithm to mine `nfer` **before** relations from historical telemetry and other real-time embedded system traces. The chapter includes case studies using data from MSL and two other datasets.

The remaining contents of the chapter are as follows. Section 2.2 provides the problem statement and motivation for this work. Section 2.3 defines the `nfer` formalism. Section 2.4 presents an algorithm for applying an `nfer` specification to an event stream. Section 2.5 describes the implementation of the system, including the external DSL. Section 2.6 illustrates the application of `nfer` to a scenario from the Mars Science Laboratory. Section 2.7 introduces modifications to the `nfer` algorithm to improve its execution time, including an experimental evaluation of their performance. Section 2.8 presents an algorithm for mining `nfer` **before** relations from historical telemetry and other real-time system traces. Section 2.9 discusses related work. Finally, Section 2.10 concludes the chapter.

2.2 Problem Statement

In this section, we briefly outline the requirements for our specification formalism. We first illustrate a concrete problem with an example. Subsequently, we outline the requirements. Consider the trace (telemetry stream) shown on the left part of Figure 2.1, that we assume has been generated by a spacecraft[2]. The trace consists of a sequence of events, or EVent Reports (EVRs) as they are named in space mission operations, each with a name, a time stamp, and a list of parameters. The events in this particular trace represent activities such as a boot process starting, a boot process ending, downlink of data to ground, and operating the antenna and radio. Our goal is to produce higher level views of this trace, which will make it easier to understand the meaning of its contents. One particular concern in this case is whether there is a downlink operation during a 5-minute time interval where the flight computer reboots twice. This scenario could cause a potential loss of downlink information. Notice the use of the term *interval*. We suggest imposing a structure on the trace, where such intervals are named and highlighted, as shown on the right part of Figure 2.1. Specifically, we want to identify the following intervals: A `BOOT` represents an interval where the flight computer is rebooting. A `DBOOT` (double boot) represents an interval where the flight computer reboots twice within a 5-minute timeframe. A `RISK` represents an interval where the flight computer reboots twice while the downlink software is also attempting to send data to Earth. Our objective now is to formalize the definition of these intervals in a specification. In this case, we need a formalism for defining the following three intervals:



NAME	TIME	PARAMS
DOWNLINK	10	size -> 430
BOOT_S	42	count -> 3
TURN_ANTENNA	80	
START_RADIO	90	
DOWNLINK	100	size -> 420
BOOT_E	160	
STOP_RADIO	205	
BOOT_S	255	count -> 4
START_RADIO	286	
BOOT_E	312	
TURN_ANTENNA	412	

Figure 2.1: An event trace and its abstractions

[2]The trace is artificially constructed to have no resemblance to real artifacts.

1. A `BOOT` interval starts with a `BOOT_S` (boot start) event and ends with a `BOOT_E` (boot end) event.

2. A `DBOOT` (double boot) interval consists of two consecutive `BOOT` intervals, with no more than 5-minutes from the start of the first `BOOT` interval to the end of the second `BOOT` interval.

3. A `RISK` interval is a `DBOOT` interval during which a `DOWNLINK` occurs.

The specification formalism should allow a user to:

1. **Define intervals** as a composition of other intervals/events. For example to define the label `BOOT` as an interval delimited by the events `BOOT_S` and `BOOT_E`, or to define a `DBOOT` to be composed sequentially of two `BOOT` intervals;

2. **Refer to time stamps** associated with events, as well as specify start and end times of generated intervals. It should be possible to define complex time constraints; and

3. **Refer to data** associated with events, as well as generate and later read data of generated intervals using a rich expression language. For example, a generated interval may have a datum value defined as the sum of two lower-level interval data.

We have found that Allen's Temporal Logic (ATL) [11], specifically its operators for expressing temporal constraints on time intervals, is a useful starting point. In ATL, a time interval represents an action or a system state taking place over a period. A time interval has a name, a start time, and an end time. ATL offers 13 mutually exclusive binary relations. Examples include: $\mathit{Before}(i, j)$ which holds iff interval i ends before interval j starts, and $\mathit{During}(i, j)$ which holds iff i starts strictly after j starts and ends before or when j ends, or i starts when or after j starts and ends strictly before j ends. An ATL formula is a conjunction[3] of such relationships, for example, $\mathit{Before}(i, j) \wedge \mathit{During}(j, k)$. A model is a set of intervals satisfying such a conjunction of constraints.

Figure 2.2 shows seven of these relations (the other six are their duals, with the dual of *equals* being identical). The temporal intervals (the boxes labeled A and B) are shown on a timeline which increases left to right. Each relation listed on the left side of the figure is demonstrated by the relative position on this timeline of the two intervals to its right.

ATL is typically used in planning for generating a plan (effectively a model) from a formula, but ATL can also be used for checking a model against a formula, as described

[3]A limited form of disjunction is also allowed but not described here.

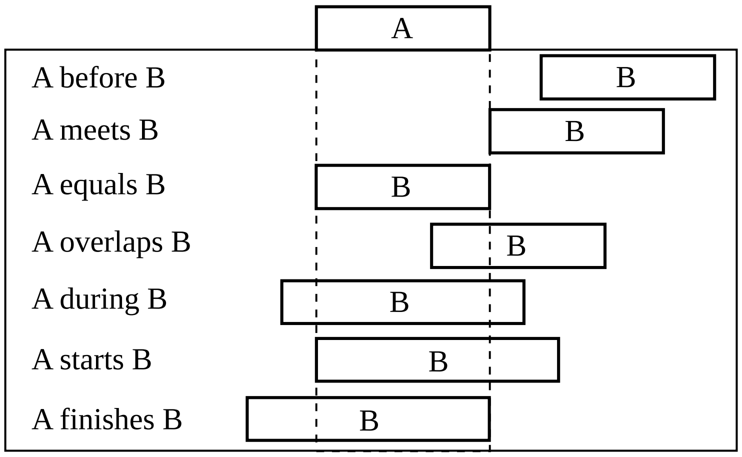


Figure 2.2: Allen's temporal relations

in [196]. Our objective is different from planning and verification. Given a trace, we want to generate a set of intervals, guided by a specification that we provide. Each interval represents an abstraction of the original trace, either of low-level events, or of other lower-level intervals. As such, the set of resulting intervals represents a hierarchical abstraction of the original trace, useful for human comprehension and further automated processing.

2.3 The `nfer` Formalism

This section describes the semantic foundations of `nfer`. The syntax given in this section forms part of the theory of `nfer`, in contrast to the domain-specific language (DSL) introduced in Section 2.5.1 that is intended for practical use. We first introduce the notion of intervals, the fundamental data structure processed by `nfer` specifications. Subsequently the core formalism is introduced including syntax and semantics, followed by derived forms which map to the core form. Finally, we present an example.

2.3.1 Intervals

A telemetry stream (for example received from a spacecraft) is a sequence of events, also referred to as a trace. In contrast to most runtime verification systems, however, the `nfer` formalism does not directly operate on such traces *from a semantics point of view.* Instead, it operates on a set of *intervals* (defined below). We will provide the definition and intuition behind intervals, and how a trace is converted into an initial set of intervals, on which `nfer` operates.

An interval represents a named section of a trace, spanning a certain time period. An interval can carry data as well, using a map. Concretely, an *interval* is a 4-tuple of the form (η, t_1, t_2, M), where $\eta \in \mathcal{I}$ is an interval name, $t_1, t_2 \in \mathcal{C}lock$ are time stamps[4] representing the start and end time of the interval, satisfying the condition $t_1 \leq t_2$, and M is a map in $\mathbb{M}$, the data that the interval carries. The interval's *duration* is $t_2 - t_1$. An interval is *atomic* if its duration is zero. The type of all intervals is denoted by $\mathbb{I}$.

A *pool* is a set of intervals, that is, an element of type $\mathbb{P} = 2^{\mathbb{I}}$. A trace τ is converted into an initial pool by a function *init* of type $\mathbb{T} \rightarrow \mathbb{P}$:

$$init(\tau) = \{\ (\eta, t, t, M)\ :\ \eta(t, M)\ \in\ \tau\ \}$$

The `nfer` system subsequently transforms this initial pool of intervals to a pool also containing the abstractions defined by the specification. We say that we are annotating the original trace with *labels* (interval names). In the following section, we illustrate how such specifications are written.

2.3.2 Syntax of the `nfer` Formalism

An `nfer` specification consists of a list of declarative *labeling* rules taking two forms: inclusive and exclusive. The application of a rule results in a set of intervals, which is the set of all possible intervals filtered to include only those that match the constraints specified by the rule.

Inclusive rules

The first form of labeling rule, called an *inclusive* rule, defines a new interval by the presence of two existing intervals:

$$\eta \leftarrow \eta_1\ \oplus\ \eta_2\ \textbf{map}\ \Phi \tag{2.1}$$

where, $\eta, \eta_1, \eta_2 \in \mathcal{I}$ are identifiers, $\oplus : \mathcal{C}lock^6 \rightarrow \mathbb{B}_2$ is a *clock predicate* on six time stamps (two for each of $\eta, \eta_1,$ and η_2), and $\Phi : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}_\perp$ is a *map function* taking two maps

[4]Time stamps have no specified units and their interpretation depends on the specification.

and returning a map or returning $\bot$, which represents non-satisfaction of a constraint on the maps. The syntax presented here contains mathematical functions to simplify the presentation.

The informal interpretation of an *inclusive* rule is as follows. Given a pool π, the rule generates a set of new intervals (a pool), each of the form (η, s, e, M), provided that in π there exist two intervals (η_1, s_1, e_1, M_1) and (η_2, s_2, e_2, M_2), such that the time constraint defined by $\oplus$ is satisfied: $\oplus(s_1, e_1, s_2, e_2, s, e)$, and such that the map function Φ produces a well-defined map as a function of the maps of the two input intervals: $M = \Phi(M_1, M_2) \neq \bot$. Note that the $\oplus$ time constraint defines the start time s and end time e of the result interval as well. Hence, one can control the time values of the generated interval.

The time constraint can, for example, express that one interval ends before the other interval starts ($e_1 < s_2$), which corresponds to one of the Allen operators. Likewise, the map function can check whether the input maps M_1 and M_2 satisfy certain conditions: if they do not, the map function returns $\bot$, but if they do, it returns a new map that is part of the generated interval. The time constraint must evaluate to true and the result of the map function must not be $\bot$ for the rule to apply.

As an example, the following rule generates an abstraction interval named `BOOT` from a `BOOT_S` (boot start) interval that occurs before a `BOOT_E` (boot end) interval, and furthermore carries the boot count contained in the `BOOT_S` interval:

$$\texttt{BOOT} \leftarrow \texttt{BOOT\_S} \;\oplus\; \texttt{BOOT\_E}\ \textbf{map}\ \Phi$$

where the two functions $\oplus$ and Φ are defined as follows:

$$\begin{array}{l} \oplus(s_1, e_1, s_2, e_2, s, e) = e_1 < s_2 \wedge s = s_1 \wedge e = e_2 \\ \Phi(m_1, m_2) = [\mathit{count} \mapsto m_1(\mathit{count})] \end{array}$$

Note how the resulting interval's start time s is constrained to be the start time of the `BOOT_S` event, and likewise the end time e is constrained to be the end time of the `BOOT_E` event. In Section 2.3.4, we introduce a pre-defined set of candidate functions for $\oplus$ inspired by Allen logic to make specifications easier to write, allowing us instead to write this rule as follows (with the same Φ function and **before** denoting the $\oplus$ function above):

$$\texttt{BOOT} \leftarrow \texttt{BOOT\_S}\ \textbf{before}\ \texttt{BOOT\_E}\ \textbf{map}\ \Phi$$

Exclusive rules

The second form of labeling rule, called an *exclusive* rule, defines an interval by the presence of one interval and the absence of a second:

$$\eta \leftarrow \eta_1 \ \textbf{unless} \ \ominus \ \eta_2 \ \textbf{map} \ \Phi \tag{2.2}$$

where, $\eta, \eta_1, \eta_2 \in \mathcal{I}$ are identifiers, $\ominus : \mathcal{C}lock^4 \rightarrow \mathbb{B}_2$ is a *clock predicate* on four time stamps (two for each of η_1 and η_2, while η is constrained implicitly), and $\Phi : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}_\perp$ is a *map function* taking two maps and returning a map or $\perp$.

The informal interpretation of an exclusive rule is as follows: Given a pool π, the rule generates a set of new intervals (a pool), each of the form (η, s_1, e_1, M_1), provided that in π there exists an interval (η_1, s_1, e_1, M_1), and there does not exist an interval (η_2, s_2, e_2, M_2), such that (i) the time constraint defined by $\ominus$ is satisfied: $\ominus(s_1, e_1, s_2, e_2)$, (ii) the map function Φ produces a well-defined map as a function of the maps of the two input intervals: $M = \Phi(M_1, M_2) \neq \perp$, and (iii) the second interval ends before the first ends $(e_2 < e_1)$.

For example, the time constraint can express that the first interval begins at the same time the second interval ends $(s_1 = e_2)$. Likewise, the map function can check whether the input maps M_1 and M_2 satisfy conditions, and return $\perp$ if not. If an η_1 interval exists, and no η_2 interval exists for which both the time constraint is true and the map function is not $\perp$, then a new η interval is generated. Unlike the first form, the start and end times and the map of the new interval cannot be controlled by the labeling rule, but are copied from the existing η_1 interval. Also unlike the first rule, the second interval must end before the first interval ends. This constraint ensures that exclusive rules are monotone – produced facts will not later be retracted.

As an example, the following rule generates an abstraction interval named `BOOT_OK` from a `BOOT` interval, if no interval named `FAILURE` with the same value of the map key *bootId* exists, that starts after the `BOOT` begins and ends before the `BOOT` ends:

$$\texttt{BOOT\_OK} \leftarrow \texttt{BOOT} \ \textbf{unless} \ \ominus \ \texttt{FAILURE} \ \textbf{map} \ \Phi$$

where the two functions $\ominus$ and Φ are defined as follows:

$$\ominus(s_1, e_1, s_2, e_2) = s_1 \leq s_2 \wedge e_2 \leq e_1$$
$$\Phi(m_1, m_2) = \textbf{if} \ m_1(bootId) = m_2(bootId) \ \textbf{then} \ [\,] \ \textbf{else} \ \perp$$

Like in the first form, we introduce a pre-defined set of candidate functions for $\ominus$ which make specifications easier to write. The above rule could be rewritten with the pre-defined function **contain** denoting the $\ominus$ function above, and the same Φ function:

$$\texttt{BOOT\_OK} \leftarrow \texttt{BOOT}\ \textbf{unless contain}\ \texttt{FAILURE}\ \textbf{map}\ \Phi$$

2.3.3 Semantics of the `nfer` Formalism

The semantics of the core form is defined in three steps: the semantics R of individual rules on pools, the semantics S of a specification (a list of rules) on pools, and finally the semantics T of a specification on traces of events. We first define the semantics of labeling rules with the *interpretation* function $R_\wp$, with the following type and definition. This function is parameterized with a *selection function* $\wp : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$, which will be explained below. Briefly stated: the Function $\wp$ is used, for example, to express the minimality constraint mentioned earlier, which helps to reduce the complexity of the algorithm introduced in Section 2.4. Let Δ be the type of rules. Semantic functions are defined using the brackets $[\![\_]\!]$ around syntax being given semantics.

```
$R_\wp[\![ \_ ]\!] \;:\; \Delta \to \mathbb{P} \to \mathbb{P}$
$R_\wp[\![ \eta \leftarrow \eta_1 \;\oplus\; \eta_2 \;\textbf{map}\,\Phi \;]\!]\; \pi \;=$
    $\textbf{let}\;\; \pi' \;=$
      $\{\; (\eta, s, e, M) \in \mathbb{I} \;\;:$
            $\exists\;\; s_1, e_1, s_2, e_2 \;\in\; \mathcal{C}lock \;\;.\;\; \exists\;\; J,K \in \mathbb{M} \;\;.$
              $(\eta_1, s_1, e_1, J) \in \pi \;\;\wedge\;\; (\eta_2, s_2, e_2, K) \in \pi \;\;\wedge$
              $\oplus(s_1, e_1, s_2, e_2, s, e) \;\;\wedge\;\; M = \Phi(J,K) \neq \bot \;\}$
    $\textbf{in}\;\; \wp\,(\pi', \pi\,)$
```

The above definition, which defines the semantics of inclusive rules, reads as follows: Given an inclusive rule $\delta \in \Delta$ and a pool π, $R_\wp[\![\delta]\!]\,\pi$ first produces a pool π' containing intervals (η, s, e, M), where there exist two intervals in π, with names η_1 and η_2, where the time constraint is satisfied, and the map resulting from applying Φ to the respective sub-maps is not $\bot$. Subsequently the *selection function* $\wp$ selects from (potentially modifies) π', informed by π as well. The selection function is said to be *idempotent* iff $\wp(\pi', \pi) = \pi'$, and a *refinement* iff $\wp(\pi', \pi) \subseteq \pi'$. The following definition gives semantics to exclusive rules:

$$
\begin{array}{l}
R_{\wp}[\![\eta \leftarrow \eta_1 \ \textbf{unless} \ominus \eta_2 \ \textbf{map}\, \Phi \ ]\!] \ \pi \ = \\
\quad \textbf{let} \ \pi' \ = \\
\quad\quad \{ \ (\eta, s_1, e_1, J) \ \in \ \mathbb{I} \ : \\
\quad\quad\quad\quad (\eta_1, s_1, e_1, J) \ \in \ \pi \ \ \wedge \\
\quad\quad\quad\quad \neg \ (\exists \ \ s_2, e_2 \ \in \ \mathcal{C}lock \ \textbf{.} \ \ \exists \ \ K \in \ \mathbb{M} \ \textbf{.} \\
\quad\quad\quad\quad\quad\quad e_2 \ < e_1 \ \wedge \ (\eta_2, s_2, e_2, K) \in \ \pi \ \ \wedge \\
\quad\quad\quad\quad\quad\quad \ominus(s_1, e_1, s_2, e_2) \ \wedge \ \Phi(J, K) \neq \bot) \ \} \\
\quad \textbf{in} \ \ \wp(\pi', \pi)
\end{array}
$$

The above definition, which defines the semantics of exclusive rules, reads as follows: Given an exclusive rule $\delta \in \Delta$ and a pool π, $R_{\wp}[\![\delta]\!]\, \pi$ first produces a pool π' containing intervals (η, s_1, e_1, J), where there exists an interval with the name η_1 in π with the same time stamps and same data, and there does not exist a second "older" $(e_2 < e_1)$ interval with the name η_2 in π, where the time constraint is satisfied, and the map resulting from applying Φ to the respective sub-maps is not $\bot$. As with inclusive rules, the selection function $\wp$ is then applied and may modify π'.

Next, we define the semantics of a list of rules, also referred to as a specification. For this we define the following one-step interpretation function S, which, given a set of rules and a pool, returns a new pool extending the input pool with added abstraction intervals resulting from taking the union of the pools generated by each rule:

$$
\begin{array}{l}
S \ [\![\ \_]\!] \ : \ \Delta^* \rightarrow \ \mathbb{P} \ \rightarrow \ \mathbb{P} \\
S \ [\![\ \delta_1 \ \ldots \delta_n \ \ ]\!] \ \pi \ \ = \pi \ \cup \ R_{\wp}[\![\delta_1 \ ]\!] \ \pi \ \ \cup \ \ldots \cup \ \ R_{\wp}[\![\delta_n \ ]\!] \ \pi
\end{array}
$$

That is, given a specification $\delta_1 \ldots \delta_n$ and a pool π, a new pool is returned by $S[\![\delta_1, \ldots, \delta_n]\!]\pi$. Finally, we define the semantics of a specification applied to a trace (a sequence of events). For this we define the interpretation function T, which, given a list of rules and a trace, returns a pool containing abstraction intervals:

$$
\begin{array}{l}
T \ [\![\ \_]\!] \ : \ \Delta^* \rightarrow \ \mathbb{T} \ \rightarrow \ \mathbb{P} \\
T \ [\![\ \delta_1 \ \ldots \delta_n \ ]\!] \ \tau \ \ = \\
\quad \textbf{least} \ \ \pi \ \ \in \ \ \mathbb{P} \ \ \textbf{such that} \ \mathit{init}(\tau) \ \subseteq \pi \ \ \wedge \ \ \pi \ \ = S \ [\![\ \delta_1 \ \ \ldots \delta_n \ ]\!] \ (\ \pi\ )
\end{array}
$$

That is, given a specification $\delta_1 \ldots \delta_n$ and a trace τ, a pool of abstractions is returned by: $T[\![\delta_1, \ldots, \delta_n]\!]\, \tau$. The resulting pool is defined as the least fixed-point of $S[\![\delta_1 \ldots \delta_n]\!] \ : \mathbb{P} \rightarrow \mathbb{P}$ that includes $\mathit{init}(\tau)$, corresponding to repeatedly applying $S[\![\delta_1 \ldots \delta_n]\!]$, starting with *init*(τ), and until no new intervals are generated. Note that the least fixed-point exists since the semantic functions are monotonic. However, our simple iterative algorithm may not reach the least fixed-point if it is an infinite set.

2.3.4 Derived Forms

As hinted at the end of Section 2.3.2, a collection of $\oplus$ functions and $\ominus$ functions have been pre-defined, along with symbols (operators) denoting them. The symbols denoting $\oplus$ functions are shown in Table 2.1 together with their definitions. Note that s_1 and e_1 are the start and end times for the left-hand interval, s_2 and e_2 are the start and end times for the right-hand interval, and s and e are the start and end times for the resulting interval. For all operators, except the **slice** operator, the start and end times of the resulting interval are the earliest and latest time stamps of the involved intervals, respectively. For the **slice** operator, the resulting time span denotes the overlapping section of two intervals. Note that the definitions of these operators differ from those of the Allen logic operators in [11], which are defined to be mutually exclusive, whereas `nfer`'s operators are not.

Table 2.1: `nfer` $\oplus$ operators

Operator $\oplus$	$\oplus(s_1, e_1, s_2, e_2, s, e)$
before	$e_1 < s_2 \wedge s = s_1 \wedge e = e_2$
meet	$e_1 = s_2 \wedge s = s_1 \wedge e = e_2$
during	$s_1 \geqslant s_2 \wedge e_1 \leqslant e_2 \wedge s = s_2 \wedge e = e_2$
coincide	$s = s_1 = s_2 \wedge e = e_1 = e_2$
start	$s = s_1 = s_2 \wedge e = \max(e_1, e_2)$
finish	$s = \min(s_1, s_2) \wedge e = e_1 = e_2$
overlap	$s_1 < e_2 \wedge s_2 < e_1 \wedge s = \min(s_1, s_2) \wedge e = \max(e_1, e_2)$
slice	$s_1 < e_2 \wedge s_2 < e_1 \wedge s = \max(s_1, s_2) \wedge e = \min(e_1, e_2)$

The informal explanation of the $\oplus$ operators is as follows: A **before** B: A ends before B starts; A **meet** B: A ends where B starts; A **during** B: all of A occurs during B; A **coincide** B: A and B occur at the exact same time; A **start** B: A starts at the same time as B; A **finish** B: A finishes at the same time as B; A **overlap** B: A and B overlap in time; A **slice** B: A and B overlap in time, and only the overlapping time span is returned.

The symbols denoting $\ominus$ functions are shown in Table 2.2 together with their definitions. Note that s_1 and e_1 are the start and end times for the left-hand interval, and s_2 and e_2 are the start and end times for the right-hand interval. Unlike the $\oplus$ operators, the $\ominus$

operators do not affect the start and end times of the resulting interval. The informal explanation of the $\ominus$ operators is as follows: A **unless after** B: A starts after B ends; A **unless follow** B: A starts where B ends; A **unless contain** B: all of B occurs during A. These operators are the *dual* of **before**, **meet**, and **during**, respectively.

Table 2.2: `nfer` $\ominus$ operators

Operator $\ominus$	$\ominus(s_1, e_1, s_2, e_2)$
after	$s_1 > e_2$
follow	$s_1 = e_2$
contain	$s_1 \leqslant s_2 \land e_2 \leqslant e_1$

The next abbreviation concerns further time constraints a user may want to impose. The core rule notation (see Section 2.3.2) allows for any time constraints to be expressed. Possible constraints include the just introduced relational operators, but also time spans, such as stating that an event B should follow an event A within 10 time units. We present the following shorthand for allowing the specification of additional time constraints in addition to the just introduced operators. Let $\odot \in$ {**before**, **meet**, **during**, **coincide**, **start**, **finish**, **overlap**, **slice**}, and let $\odot_p$ denote the corresponding clock predicate. The following derived rule form:

$$\eta \leftarrow \eta_1 \;\; \odot \;\; \eta_2 \; \textbf{within} \; \Theta \; \textbf{map} \; \Phi$$

where $\Theta : \mathcal{Clock}^6 \rightarrow \mathbb{B}_2$ is a predicate on six time stamps, is synonymous with:

$$\eta \leftarrow \eta_1 \; (\odot_p \land \Theta) \; \eta_2 \; \textbf{map} \; \Phi$$

We shall allow the time constraint (**within**) and/or map transformation (**map**) to be left out, in which case they assume the default function values respectively $\lambda s_1, e_1, s_2, e_2, s, e.\mathit{true}$ and $\lambda m_1, m_2.\,[\,]$.

So far rules can only be defined that refer to one operator and one additional clock predicate as shown above. This format presents a simple notation with a clean semantics. However, further convenient syntax allows rules containing more than one operator, for example: $A \;\leftarrow\; (B$ **before** $C)$ **overlap** D. Such rules are mapped into multiple rules in the core form (in this case two). The external DSL described in Section 2.5.1 allows such enriched rules.

2.3.5 Example

As an example, we will formalize the three rules that were informally stated in Section 2.2. The specification similarly consists of three rules:

$\texttt{BOOT} \leftarrow \texttt{BOOT\_S}$ **before** $\texttt{BOOT\_E}$ **map** $(\lambda\ m_1,m_2\ .\ [\mathit{count} \mapsto m_1(\mathit{count})])$

$\texttt{DBOOT} \leftarrow \texttt{BOOT}$ **before** $\texttt{BOOT}$ **within** $(\lambda\ s_1,e_1,s_2,e_2,s,e\ .\ e{-}s \leqslant 300)$ **map** *snd*

$\texttt{RISK} \leftarrow \texttt{DOWNLINK}$ **during** $\texttt{DBOOT}$ **map** *snd*

The rules should be mostly self-explanatory (time is assumed measured in seconds). The first rule creates from the two sub-maps m_1 and m_2 a new map, mapping count to the same value as in m_1. The function *snd* selects m_2 from a binary tuple (m_1, m_2).

Let us illustrate how this specification is evaluated on the trace in Figure 2.1. This trace is first converted into an initial pool. The semantic S function on (page 15) will go through three iterations when applied to this initial pool before a fixed-point is reached. The added intervals in each iteration are as follows, assuming the selection function is idempotent:

$$\begin{array}{lll}
1 & : & \{\ (\text{BOOT}, 42, 160, [\text{count} \mapsto 3]), \\
 & & \phantom{\{}\ (\text{BOOT}, 255, 312, [\text{count} \mapsto 4]), \\
 & & \phantom{\{}\ (\text{BOOT}, 42, 312, [\text{count} \mapsto 3])\ \} \\
2 & : & \{\ (\text{DBOOT}, 42, 312, [\text{count} \mapsto 4])\ \} \\
3 & : & \{\ (\text{RISK}, 42, 312, [\text{count} \mapsto 4])\ \}
\end{array}$$

Note that the third interval in step one, $(\mathsf{BOOT}, 42, 312, [\mathsf{count} \mapsto 3])$, is irrelevant, since it spans other BOOT intervals. In the next section, we will define the concept of *minimality* to restrict the generated intervals to only those in which we are interested.

2.4 Monitoring Algorithm

The semantics given in Section 2.3 is expressed using an interpretation function $R_{\wp}[\![\delta]\!]\ \pi$ that operates on a finite pool, built from a finite, known trace. However, in online telemetry stream analysis, the stream of incoming events is, in theory, infinite, since we do not know when it ends. Therefore, constructing a pool from such a trace is not practical. To interpret an `nfer` specification with respect to a live telemetry stream (online), events in the stream must be converted to intervals and processed one at a time as they arrive. Algorithm 1 expresses a simple procedure for interpreting one interval at a time, either coming from the trace, or produced by the algorithm. The algorithm is defined as a function, calling itself recursively with newly produced intervals. An informal explanation of Algorithm 1 follows with a detailed example in Section 2.4.1.

Rules are assumed to be in a simplified binary form only referring to one temporal operator. Any specification can be rewritten into a semantically equivalent one consisting only of such binary rules, and it simplifies the algorithm. In a rule of the form $\eta \leftarrow \eta_1 \oplus \eta_2$ **map** Φ we refer to η as the *rule head*, and to $\eta_1 \oplus \eta_2$ **map** Φ as the *rule body*, or the *rule expression*. In such a rule body, we refer η_1 as the left-hand label, and η_2 as the right-hand label. We refer to the head, body, and left and right labels of exclusive rules in the same way. For each rule, the algorithm keeps track of three sets of already produced intervals relevant for that rule: *rule.LeftCache* holds the intervals which have been produced and matched the left-hand label of the rule, *rule.RightCache* holds the intervals which have been produced and matched the right-hand label of the rule, and finally *rule.Produced* holds the intervals which have been produced by the rule itself. In addition, the algorithm uses a variable *Subscribers*, which maps interval names to those rules that subscribe to intervals with those names.

Each rule also has methods that behave according to the operators $\oplus$ and $\ominus$, and map function Φ. Method *rule.testInclusion* checks that the time operator $\oplus$ is true and that the map function Φ does not return $\bot$. Method *rule.testExclusion* checks that the time operator $\ominus$ is true and that the map function Φ does not return $\bot$. Finally, method *rule.createInterval* generates a new interval using the time operator and map function.

An informal explanation of Algorithm 1 follows. On Line 2, the procedure accesses the map *Subscribers* that associates interval names with rules. The rules that subscribe to the submitted interval's name are then iterated over.

Between lines 4 and 14, the algorithm handles the case where the passed interval name matches the left-hand label of the rule. If the rule is an exclusive rule (see Section 2.3.2), then *rule.RightCache* is iterated over looking for any intervals for which *rule.testExclusion* is true. If no such interval is found, then a new interval is generated and added to the set *New*. If the rule is an inclusive rule (see Section 2.3.2), then *rule.RightCache* is iterated over looking for any intervals for which *rule.testInclusion* is true. If such an interval is found, then a new interval is generated and added to the set *New*.

Between lines 15 and 18, the algorithm handles the case where the submitted interval

Algorithm 1 Basic `nfer` Processing Algorithm

```
procedure PROCESS(interval)
    for rule $\in$ Subscribers[interval.name] do
        New $\leftarrow \varnothing$
        if interval.name = rule.leftLabel then
            if rule is *exclusive* then
                exclude $\leftarrow$ *false*
                for rightIntv $\in$ rule.RightCache do
                    exclude $\leftarrow$ exclude $\vee$ rule.testExclusion(interval, rightIntv)
                if $\neg$ exclude then
                    New $\leftarrow$ New $\cup$ {rule.createInterval(interval)}
            else
                for rightIntv $\in$ rule.RightCache do
                    if rule.testInclusion(interval, rightIntv) then
                        New $\leftarrow$ New $\cup$ {rule.createInterval(interval, rightIntv)}
        if interval.name = rule.rightLabel $\wedge$ rule is *inclusive* then
            for leftIntv $\in$ rule.LeftCache do
                if rule.testInclusion(leftIntv, interval) then
                    New $\leftarrow$ New $\cup$ {rule.createInterval(leftIntv, interval)}
        if interval.name = rule.leftLabel then
            rule.LeftCache $\leftarrow$ rule.LeftCache $\cup$ {interval}
        if interval.name = rule.rightLabel then
            rule.RightCache $\leftarrow$ rule.RightCache $\cup$ {interval}
        selected $\leftarrow$ select(New, rule.Produced)
        for new $\in$ selected do
            rule.Produced $\leftarrow$ rule.Produced $\cup$ {new}
            selected $\leftarrow$ selected $\cup$ PROCESS(new)
    return selected
```

name matches the right-hand label of the rule, and the rule is inclusive. In such a case, *rule.LeftCache* is iterated over looking for any intervals for which *rule.testInclusion* is true. If such an interval is found, then a new interval is generated and added to the set *New*. No work is required if the rule is exclusive, other than adding the interval to *rule.RightCache*.

Between lines 19 and 22, the algorithm adds the submitted interval to either or both caches, depending on which labels the interval name matches. It is necessary to wait to add the interval to the caches until after all of the condition tests are performed so that the interval cannot be matched against itself.

On Line 23 the selection function ($\wp$ in the semantics in Section 2.3.3) is applied and its results are iterated over on Line 24. Each interval in the selected set is added to the *rule.Produced* set, and the PROCESS function is called on the interval recursively. The results are added to the selected set then returned.

2.4.1 Example

This section presents an example illustrating an execution of Algorithm 1. Assume the following rule: `BOOT` ← `BOOT_S` **before** `BOOT_E`. We will trace the processing of two intervals: (`BOOT_S`, 10, 10, [ ]) and (`BOOT_E`, 20, 20, [ ]).

First, PROCESS((`BOOT_S`, 10, 10, [ ])) is called. The above rule is found to be a subscriber to this interval on Line 2 because its label (BOOT_S) is referenced in the rule's expression. The condition on Line 4 is true because BOOT_S is used on the left side of the **before** operator in the rule expression. The rule is inclusive, so the condition on line 5 is false. Since the condition was false, execution continues on Line 12 by iterating over the rule's *RightCache*, which is empty. The condition on Line 15 is false, since the interval's name (BOOT_S) does not appear on the right side of the **before** operator. Execution continues on Line 19, where the condition is met and so the interval is added to the rule's *LeftCache*. The condition on Line 21 is not met, so execution continues on Line 23. The *select* function is called on $(\varnothing, \varnothing)$, and the results are iterated over. If the select function is a *refinement* (see Section 2.3.3), then the returned set will also be empty, and the procedure returns.

Next, PROCESS((`BOOT_E`, 20, 20, [ ])) is called. The same rule is found to be a subscriber because BOOT_E is referenced in the rule's expression. Since BOOT_E appears on the right side of the **before** operator, the condition on Line 4 is false, but the condition on Line 15 is true and execution continues on Line 16. The rule's *LeftCache* contains the BOOT_S interval from above, so *rule.testInclusion* is called on the two intervals. The *testInclusion* method returns true, since the conditions of the **before** operator are met and there is no map function, so a new interval is created and added to the *New* set. The condition on

Line 19 is false, but the condition on Line 21 is true, so the interval is added to the rule's *RightCache*. Next, the *select* function is called on two arguments: *New*, which contains the created interval; and the set of the rule's previously produced intervals, which is empty. If the *select* function returns a set containing the created interval, then that interval is added to the *Produced* set and then PROCESS is called on it recursively.

2.4.2 Complexity

When an `nfer` specification contains circular references, Algorithm 1 may not terminate. A circular reference occurs when rules may be composed in such a way that the same label appears on both the left and right sides of the $\leftarrow$. An example of such a rule is `A` $\leftarrow$ `A` **coincide** `A`. With an idempotent selection function, infinite `A` intervals will be generated if an interval such as $(\mathtt{A}, 10, 10, [\,])$ appears twice. Note that the minimality selection function (described below) removes many, but not all, results that would otherwise result in infinite recursion.

Circular references to not guarantee infinite recursion, however, and they are often useful to describe recurring behavior. As a result, we do not prohibit them in general but only warn that they may lead to non-termination.

The asymptotic complexity of Algorithm 1 for specifications without circular references is $O(n^3)$ in the length of the trace. We assume that the methods associated with a rule (*testInclusion*, *testExclusion*, *createInterval*) are constant time expressions. Given the number r of rules in a specification, and the number n of intervals in a trace, we can find the complexity as follows, with n_x referring to n's value ($n_x = n$, where x differentiates its use in the algorithm).

$$n_{trace} \times r \times \mathbf{max}\left(\underbrace{n_{\text{right}} + s + (1 \times n_{\text{recurse}})}_{\text{exclusive}},\ \underbrace{n_{\text{right}} + n_{\text{left}} + s + ((n_{\text{left}} + n_{\text{right}}) \times 2n_{\text{recurse}})}_{\text{inclusive}}\right)$$

For each interval in the trace (n_{trace}) (Line 1), for each rule (r) (Line 2), we calculate the maximum of the two cases of: an exclusive rule versus an inclusive rule. Assume first that the rule is *exclusive*. If the submitted interval name matches the left-hand label of the rule, check all the intervals in the right cache (n_{right}), and generate at most one interval (Line 7). Call the selection function (s) (Line 23), and for each interval returned by the selection function, call PROCESS recursively (Line 26). If the selection function is a refinement, then the number of iterations in the loop on Line 23 is bounded by the cardinality of the set *New*, which in this case is 1 (if the selection function is not a refinement, then the complexity of Algorithm 1 is unbounded). On the other hand, if the submitted interval name matches the right-hand label of the rule the interval is just stored (not included in the complexity calculation).

Assume next that the rule is *inclusive*. If the submitted interval name matches the left-hand label of the rule, for all the intervals in the right cache (n_{right}), generate an interval (Line 12). Symmetrically, if the submitted interval name matches the right-hand label of the rule, for all the intervals in the left cache (n_{left}), generate an interval (Line 16). Subsequently the selection function is called (s). The cardinality of *New* is at most $n + n$ since an interval may be created for each pairing of the newly submitted interval with all the intervals in both the left and right cache. For each interval returned by the selection function, call PROCESS recursively (Line 26).

The complexity of the algorithm is in part determined by the complexity of the selection function and the cardinality of the set it returns. In the case of an idempotent selection function, its complexity should be constant and the cardinality of the returned set should be at most $2n$. As long as the complexity of the selection function is not super-linear, the complexity of the processing function is $O(n^3)$ in the length of the trace, which appears impractical.

2.4.3 Minimality

This complexity is largely related to the number of new intervals returned by the selection function. Limiting the size of this set is therefore desirable as long as it is consistent with pragmatic needs. In practice, it is typically not desirable to use an idempotent selection function and return every matching interval. This is similar to how practical implementations of regular expressions use greedy matching instead of complete matching [135]. In the example in Section 2.3.5, three `BOOT` intervals are generated, but only two of them are relevant. The interval that begins at time 42 and ends at time 312 does not represent an intended abstraction, i.e. a period when the spacecraft is continuously booting. Instead, it contains two smaller `BOOT` intervals with a gap between where there is no boot taking place.

We observe that the property that differentiates relevant intervals from others is their *minimality*. An interval is defined as minimal if no other interval with the same label occurs **during** it. That is, given a pool $\pi \in \mathbb{P}$ and an interval (η, s, e, M),

$$\text{minimal}(\eta,s,e,M)(\pi) \leftarrow \nexists\ (\eta',s',e',M') \in \pi\ .\ \eta = \eta' \wedge\ s \leq s' \wedge\ e' \leq e$$

The following selection function implements the minimality constraint. The function is a refinement and has a complexity determined by the cardinality of the *New* and *Prior* sets. In Section 2.7 we explore methods to limit the size of these sets.

Minimal Selection Function

```
1: procedure SELECT($New$, $Prior$)
2:     $\{i \in New : \nexists\ j \in Prior \cup New \setminus \{i\}\ .\ i.name = j.name \wedge i.start \leq j.start \wedge j.end \leq i.end\}$
```

2.5 Implementation

The `nfer` logic has been implemented as a shallow, internal DSL (iDSL - essentially a library where functions on data are coded as functions in the implementation language), as well as an external DSL (eDSL - a stand-alone domain-specific language) in both Scala and C. The Scala iDSL was described in detail in [129] but is omitted here. This section provides an overview of the eDSL as well as its implementation in both Scala and C. The C implementation is available to the public under the terms of the Gnu Public License version 3 (GPLv3) at `http://nfer.io`. The C version, furthermore, supports R [220] and Python [224] language interfaces.

Each kind of DSL (internal and external) has advantages and disadvantages. The advantages of the iDSL are ease of implementation and modification, ease of use in already existing programming environments (like IDEs), as well as a maximally expressive formalism for writing arbitrary data processing functions to be called in specifications. The disadvantages include the requirement that the user must be a programmer in the host language (Scala or C), generally a somewhat poorer syntax compared to an external DSL, and difficulty in analyzing specifications (in the case of a *shallow* internal DSL, where the host language forms a fundamental part of the DSL). Due to these negative iDSL properties, we chose to implement an eDSL. The choice of iDSL versus eDSL in a practical situation depends on the value given to the advantages and disadvantages mentioned just above.

In addition to these textual languages, we have experimented with visual entering of rules. A prototype GUI has been designed and implemented[5] for visual entering of rules

[5]The GUI was designed and implemented by Nathaniel Guy (JPL).

based on an initial visualization of a trace. That is, the user is in this system presented a linear visualization of a trace, and can interact with this by selecting events of importance (point-and-click), thereby informing the system of the event patterns forming the body of a rule. The rationale behind this approach is the acknowledgement that it can be difficult for users to write rules without some guidance based on the format of actual traces. This prototype forms a basis for future work, and is not elaborated on further in this thesis.

This section first introduces the eDSL syntax, then gives an overview of the actor-based Scala implementation, then describes the C implementation, and finally describes the R and Python interfaces.

2.5.1 The External DSL

This section introduces the external DSL (eDSL) for writing `nfer` specifications. Consider the double boot example written in the `nfer` notation in Section 2.3.5. This example can be written as follows in the external DSL:

$$\mathsf{BOOT} :- \mathsf{BOOT\_S}\ \textbf{before}\ \mathsf{BOOT\_E}\ \textbf{map}\ \{\mathsf{count} \rightarrow \mathsf{BOOT\_S.count}\}$$

$$\begin{aligned}&\mathsf{DBOOT} :- \mathsf{b1{:}BOOT}\ \textbf{before}\ \mathsf{b2{:}BOOT}\\ &\quad \textbf{where}\ \mathsf{b2}.\textbf{end} - \mathsf{b1}.\textbf{begin} \leq 300\ \textbf{map}\ \{\mathsf{count} \rightarrow \mathsf{b1.count}\}\end{aligned}$$

$$\mathsf{RISK} :- \mathsf{DOWNLINK}\ \textbf{during}\ \mathsf{DBOOT}\ \textbf{map}\ \{\mathsf{count} \rightarrow \mathsf{DBOOT.count}\}$$

The syntax should be self explanatory except for a few details. The first rule creates a BOOT interval containing a data map, which maps count to the count value of the BOOT_S interval. This illustrates how data of particular intervals can be referenced. In cases where a rule body contains more than one occurrence of the same interval, as BOOT in the second rule, these can be labelled (here b1 and b2) to enable reference to their data and begin and end time points. The second rule illustrates a **where**-clause expressing a constraint on time values[6]. Such constraints can also refer to data. An expression language covering Boolean expressions involving arithmetic operations and comparisons (integers and real numbers) as well as string comparisons is built in.

A grammar for the eDSL is shown in Figure 2.3. A specification is either a list of rules or a list of modules. Modules are useful for conceptually grouping a large number

[6]Note that the eDSL uses **begin** to denote the start time of an interval, in contrast to the notation in Section 2.3 where it was referred to as *start* (time).

$\langle \mathit{spec} \rangle ::= \langle \mathit{rule} \rangle^{*} \mid \langle \mathit{module} \rangle^{*}$

$\langle \mathit{module} \rangle ::=$ **module** $\langle \mathit{id} \rangle$ ‘`{`’ [**import** $\langle \mathit{id} \rangle^{*,}$ ‘`;`’] $\langle \mathit{rule} \rangle^{*}$ ‘`}`’

$\langle \mathit{rule} \rangle ::= \langle \mathit{id} \rangle$ ‘`:-`’ $\langle \mathit{interval} \rangle$ [$\langle \mathit{whereExp} \rangle$] [$\langle \mathit{mapExp} \rangle$] [$\langle \mathit{endPoints} \rangle$]

$\langle \mathit{interval} \rangle ::= \langle \mathit{intervalPrim} \rangle$ ($\langle \mathit{op} \rangle$ $\langle \mathit{intervalPrim} \rangle$)*

$\langle \mathit{intervalPrim} \rangle ::=$ [$\langle \mathit{id} \rangle$ ‘`:`’] $\langle \mathit{id} \rangle$ | ‘`(`’ $\langle \mathit{interval} \rangle$ ‘`)`’

$\langle \mathit{whereExp} \rangle ::=$ **where** $\langle \mathit{exp} \rangle$

$\langle \mathit{mapExp} \rangle ::=$ **map** ‘`{`’ ($\langle \mathit{id} \rangle$ ‘$\rightarrow$’ $\langle \mathit{exp} \rangle$)*, ‘`}`’

$\langle \mathit{endPoints} \rangle ::=$ **begin** $\langle \mathit{exp} \rangle$ **end** $\langle \mathit{exp} \rangle$

$\langle \mathit{op} \rangle ::=$ **also** | **before** | **meet** | **during** | **coincide** | **start** | **finish** | **overlap**
| **slice** | $\langle \mathit{exclude} \rangle$

$\langle \mathit{exclude} \rangle ::=$ **unless** (**after** | **follow** | **contain**)

$\langle \mathit{exp} \rangle ::=$ ... | $\langle \mathit{id} \rangle$ ‘`.`’ ($\langle \mathit{id} \rangle$ | **begin** | **end**) | $\langle \mathit{id} \rangle$ ‘`(`’ $\langle \mathit{exp} \rangle^{+,}$ ‘`)`’ | ...

Figure 2.3: Grammar for external `nfer` DSL

of rules. A module can import other modules and contains rules. The last occurring is the main module. A rule body is defined by an interval expression (interval), and three optional items: a where-constraint, a map, and a definition of the end points *begin* and *end*, in case the default generated time points are not desired. An interval (expression) is a composition of primary intervals separated by the temporal operators. A primary interval consist of an optional label, and an interval name. Alternatively a primary interval can be an interval in parentheses. Value expressions are standard and only specified partially here, focusing on the syntax for referring to data fields, and begin and end times of intervals; as well as function calls. Function calls are calls to user-defined functions in a programming language, registered to the monitor. This allows a user to call a function in, for example, Python or Scala, achieving the full expressiveness of a real programming language. Among the temporal operators is one that has not mentioned before: **also**, representing the lack of any constraints at all. This operator allows for time constraints defined purely using the **where** clause.

2.5.2 Scala Implementation

The Scala implementation is based on Akka actors communicating via asynchronous message passing through a publish/subscribe model built with Apache Kafka [138]. Each rule in an `nfer` specification results in an actor, which subscribes to intervals referenced by name in the body of the rule, and publishes the interval mentioned in the rule head (head, in head :− body) to the shared bus. This means that rule actors are only passed intervals which are pertinent to their execution. Figure 2.4 shows the `nfer` implementation's internal configuration corresponding to the double boot example in Section 2.5.1. The Kafka publish/subscribe framework is represented in the center by the Shared Telemetry Bus. Each actor is represented by a circle, with arrows showing the messages that are passed to the actor (those it subscribes to), as well as the messages the actor publishes back.

For example, the RISK actor subscribes to both `DBOOT` and `DOWNLINK` intervals, and publishes back `RISK` intervals. A special actor receives messages from the spacecraft and publishes them to the bus. When a rule actor publishes an interval, any subscribers will be notified and can build on this interval to create yet new intervals. The `nfer` formalism is declarative and the order in which rules are declared is unimportant. Likewise, the order in which actors execute is also unimportant, since the results of one actor cannot inhibit the behavior of any other actor.

The implementation can process events *online*, as they come down to the ground from the spacecraft, or it can process a log of events stored on a file system. When processing logs, ground operators are usually only interested in recent events. However, there can be a need to analyze the telemetry stream from earlier points in time stored as multiple logs, e.g. from the start of the mission. In this case, it is not expedient to process all events in the full telemetry stream from the start of the mission whenever the `nfer` system is activated. Instead, `nfer` can be used to incrementally create intervals from older logs, which can then be stored for later use as an abstraction of those logs. In other words: stored intervals produced by `nfer` represent abstractions of the past.

Internal Representation of Rules in Scala

As already mentioned, each rule is implemented as an actor receiving and publishing events to the event bus. The rule inside an actor is represented by a tree structure closely corre-

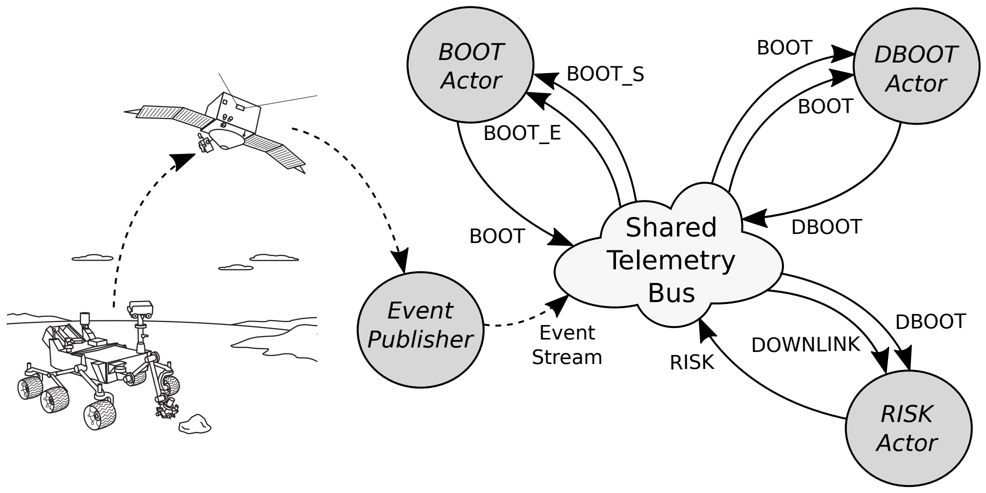


Figure 2.4: Actor network corresponding to double boot example in Section 2.5.1

sponding to the abstract syntax tree obtained by parsing the body of the rule. Each node in the tree corresponds to a temporal operator, or a leaf node representing intervals to which the actor subscribes. During execution, a node contains the intervals that have thus far matched the corresponding sub-expression. To illustrate this tree structure, consider a slightly different formulation of the last two rules DBOOT and RISK above, merging them into one:

RISK :− DOWNLINK **during** (b1:BOOT **before** b2:BOOT)
  where b2.**end** − b1.**begin** ≤ 300 **map** {count → b1.count}

The body of this rule contains two temporal operators. This rule is represented as the tree shown in Figure 2.5. Each node lists (in the top part of the box) a node number, a (possibly auto-generated) name, and an indication of which temporal operator it represents, including "Atomic" for the leaf nodes. The tree is shown after the following three intervals have been submitted: (BOOT_S, 100, 100, [count ↦1]), (BOOT_E, 200, 200, [ ]), and (DOWNLINK, 300, 300, [ ]). As can be seen, some of the nodes contain intervals, and others do not yet. This reflects the step-wise evaluation of the rule as intervals arrive. An interval is submitted to the appropriate atomic leaf nodes of the tree, and then ascends the tree. It merges with other intervals according to the temporal operators, until the top node is reached and an interval is generated and published on the bus if the constraint is satisfied.

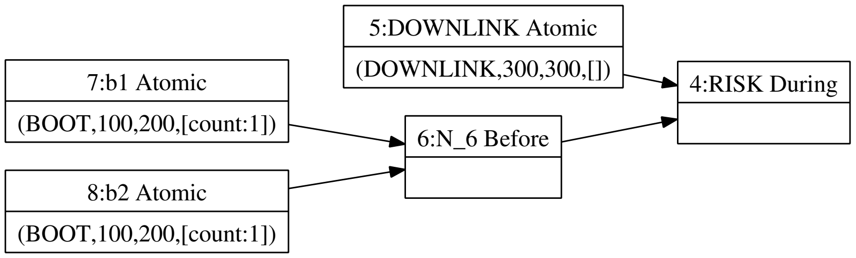


Figure 2.5: Tree structure representing rule `RISK`

An interesting observation is that this data structure resembles the Rete data structure [92] used in rule-based systems, and explained in detail in [73]. In [107] we implemented the Rete algorithm following [73], while augmenting it for runtime verification. For a mention of other rule-based systems, the reader is referred to Section 2.9. In rule systems, rules typically have the form $c_1, \ldots, c_n \rightarrow a$, representing an action a to be executed when conditions $c_1, \ldots, c_n$ are true. Conditions and actions refer to facts stored in the Rete network, similar to how intervals are stored in the rule trees. The Rete algorithm maintains a single directed acyclic graph of nodes containing produced facts. The single graph represents all the rules in the rule program, and allows rules to share parts of the graph, thus reducing the amount of evaluations needed. In `nfer`, each rule is represented by its own tree, not shared with other rules. Similar to the `nfer` algorithm, the Rete algorithm for each node handles data coming up from a child node (left or right) by traversing the "other" child-node for matches. Whereas the `nfer` tree structure consists of nodes all of the same kind, the Rete data structure involves four kinds of different nodes.

2.5.3 C Implementation

The C language implementation is single threaded and conforms closely to the algorithm in Section 2.4. It encodes all rules as relations between two intervals[7]. Each rule subscribes to a left and a right label. For example, the following rule subscribes to BOOT_S as its left label and BOOT_E as its right label because of their position relative to the **before** operator:

BOOT :– BOOT_S **before** BOOT_E **map** {count → BOOT_S.count}

[7]The eDSL supports unary rules but we avoid describing them here as they represent a special case.

The implementation keeps two linked lists of subscribers to every label, one for left labels and one for right labels. When an interval is received, both lists of subscribers are iterated over for that label. Separate left and right lists are necessary because each rule may subscribe to two different labels, so it may appear in two different linked lists.

Nested rules are handled by adding anonymous, internal intervals to which other rules subscribe. All rules must have a label for the intervals they create, so nested rules create anonymous intervals with generated labels. Uniqueness is guaranteed for these labels by using an augmented naming alphabet and they are omitted from the final output of the algorithm. The parent rule that relies on the results of a nested rule then subscribes to this generated label. For example, the nested rule in Section 2.5.2 becomes the following two rules in the C implementation:

$BOOTBOOT1 :– b1:BOOT **before** b2:BOOT
  where (b2.**end** – b1.**begin** $\leq$ 300)
  map { $count1 $\rightarrow$ (b1.count) }

RISK :– DOWNLINK **during** $BOOTBOOT1
  map { count $\rightarrow$ ($BOOTBOOT1.$count1) }

To ensure the correct behavior in **where**, **map**, **begin**, and **end** expressions, data items must be renamed and passed between nested rules. The external DSL only allows such expressions to be specified at the highest level, but they may refer the intervals in a nested rule. In the example above, the RISK interval sets the map item “count” to be equal to the value of the “count” data item of the left BOOT interval in the nested rule. The value is copied to an intermediate map key, “$count1”, of the generated interval $BOOTBOOT1 so it can be accessed in the RISK rule.

Similarly, nested rules must inherit parts of **where** expressions that apply to them. This is important due to the influence of selection functions on the result. In the example above, the **where** expression is applied to the nested rule (b1:BOOT **before** b2:BOOT) because it applies only to the intervals in that rule. If the nested rule does not contain the restriction (b2.**end** – b1.**begin** $\leq$ 300), then the wrong intervals may be selected. If a subexpression of a **where** restriction concerns only a nested rule, the subexpression will be applied to the nested rule and replaced in the original expression with a generated map value.

The C implementation includes some performance optimizations. Strings are interned in dictionaries and expressions are stored and processed using Reverse Polish Notation

(RPN). It is meant as a reference implementation, but its execution time and memory requirements are low enough to be used in an embedded setting.

2.5.4 R and Python Integrations

The C implementation of `nfer` includes two integrations with programming languages. The R integration allows a programmer to mine rules from event data and to apply those and other rules to transform traces. In addition to those capabilities, the Python integration allows programs to be automatically instrumented and includes a graphical output. Both integrations are meant to reduce the barriers an engineer may encounter when adding `nfer` to an existing project.

R

R is an interpreted programming language especially popular with statisticians and scientists [220]. Unlike most programming languages, R has no scalar types, preferring to treat data as vectors. R supports user packages and has a large community of contributors.

The `nfer` R Application Programming Interface (API) is simple and revolves three primarily operations: loading rules, mining rules, and applying rules. The example in Figure 2.6 loads the R language interface on Line 1, reads a specification from a file on Line 2, and applies it to a dataframe of events on Line 4. This returns a dataframe of intervals produced by applying the specification. In the example, the *summary* function is applied to this structure on Line 5.

The R API also supports mining rules from data using the algorithm described in Section 2.8. The function that mines rules returns a handle that can be applied to a dataframe to produce intervals, just like loading a specification. In Figure 2.7, the mining algorithm is called on Line 2 and then applied to the dataframe of events on Line 3.

The R API contains only batch operations, and this is well suited for how R is typically used as a data analysis tool. Because the API uses standard R dataframes, it integrates easily into other R programs. The API uses the C implementation of the `nfer` processing and mining algorithms, so they are much faster than if the algorithms were written natively in R.

```
> source("R/nfer.R")
> handle <- Rnfer("examples/specs/ssps.nfer")
> data <- read.table("examples/logs/ssps.events", sep="|", header=FALSE)
> intervals <- RnferApply(handle, data)
> summary(intervals)
     name                start                 end
 Length:19342       Min.   :8.238e+05   Min.   :1.080e+09
 Class :character   1st Qu.:2.452e+13   1st Qu.:2.455e+13
 Mode  :character   Median :4.023e+13   Median :4.024e+13
                    Mean   :4.460e+13   Mean   :4.462e+13
                    3rd Qu.:6.589e+13   3rd Qu.:6.592e+13
                    Max.   :9.335e+13   Max.   :9.335e+13
```

Figure 2.6: Example of using the `nfer` R API to read and apply rules

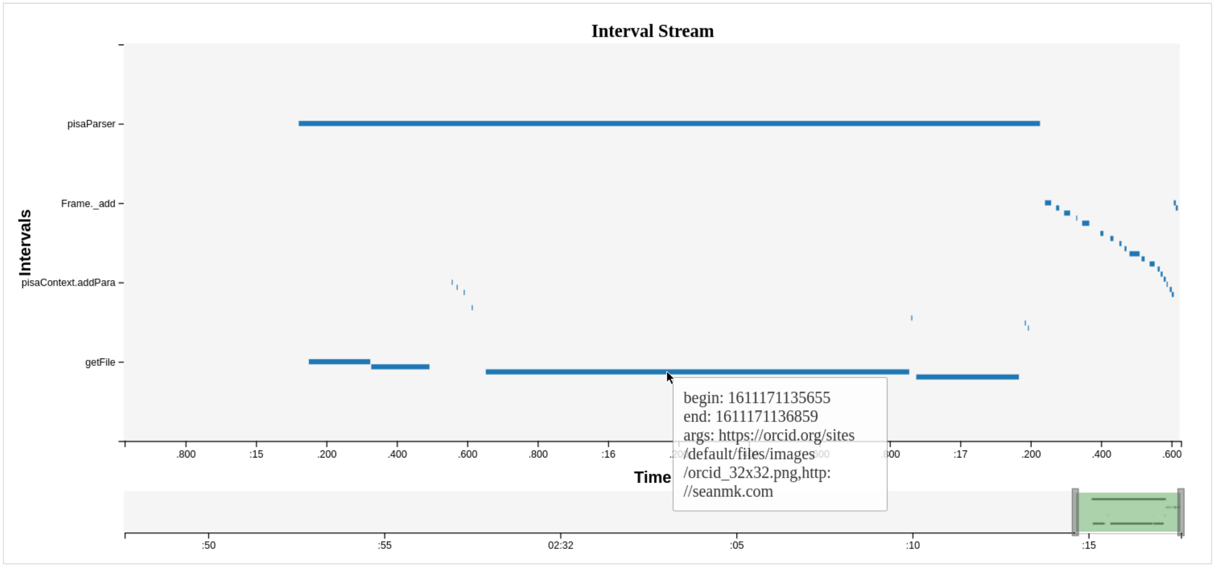
```
> df <- read.table("examples/logs/ssps.events", sep="|", header=FALSE)
> learned <- RnferLearn(df)
> intervals <- RnferApply(learned, df)
> summary(intervals)
     name                start                 end
 Length:4864        Min.   :8.238e+05   Min.   :1.080e+09
 Class :character   1st Qu.:2.289e+13   1st Qu.:2.290e+13
 Mode  :character   Median :3.948e+13   Median :3.948e+13
                    Mean   :4.475e+13   Mean   :4.475e+13
                    3rd Qu.:6.771e+13   3rd Qu.:6.772e+13
                    Max.   :9.335e+13   Max.   :9.335e+13
```

Figure 2.7: Example of using the `nfer` R API to learn and apply rules

Python

Python is an interpreted programming language popular with data scientists but also with the wider programming community [224]. It is considered a good pedagogical language due to its high level constructs and concise syntax. Python also has an enormous module library and is easy to extend. Telemetry analysis scripts at JPL often use Python.

The `nfer` Python API supports similar usage to the R API but also includes streaming support and the ability to instrument Python programs automatically and to display their timings in a streaming graphical web interface. Figure 2.8 shows the web interface displaying some selected functions from an instrumented copy of the open source program `xhtml2pdf` [112]. The program was called with arguments to download and convert the author's personal website to a pdf. In the figure, the solid blue lines represent the interval when the function on the vertical axis executed. The user can mouse-over an interval to learn its precise begin-and-end-times and, in the case of a function, its arguments. For example, the **getFile** function executed from `:15.655` until `:16.859` and had two arguments: `https://orcid.org/sites/default/files/images/orcid_32x32.png`, and `http://seanmk.com`.



begin: 1611171135655
end: 1611171136859
args: https://orcid.org/sites
/default/files/images
/orcid_32x32.png,http:
//seanmk.com

Figure 2.8: The `nfer` Python API graphical interface displaying selected functions from an instrumented copy of xhtml2pdf

Instrumenting a program with the `nfer` Python API is simple and requires only a single function call. In the example below, the `xhtml2pdf` module is instrumented on Line 6. After instrumentation, rules may be added with an internal DSL (shown on Line 7) and the Graphical User Interface (GUI) may be started via a non-blocking call (Line 8).

```
1  from nfer.instrument import instrument
2  from nfer.rules import when
3  from nfer.gui import gui
4  from nfer.nfer import monitor
5
6  instrument( "xhtml2pdf" )
7  monitor( when("execute").during("command").name("pisa") )
8  gui( intervals =["getFile","pisaContext.addParam","Frame.__add","pisaParser"] )
```

Figure 2.9: Example of using the `nfer` Python API to instrument a program and launch the GUI

Like the R API, the Python API uses the C implementation of the `nfer` processing and mining algorithms. Unlike the R API, however, the Python API contains a rich internal DSL for instantiating rules and executes `nfer` in a separate thread from the main application.

2.6 Example Application to Warning Analysis

As noted earlier, the `nfer` tool has been applied to processing of telemetry from the Curiosity rover. In this section, we briefly describe an application to a task that is traditionally performed either manually or by ad-hoc scripts. We consider the problem of automatically labeling warning messages that are *anticipated* due to known idiosyncrasies of the system, and therefore can be ignored. Events (EVRs) produced by Curiosity are associated with a *severity* level, which is used to distinguish between expected and unexpected behavior. One of the severity levels is *WARNING*, which indicates potentially anomalous behavior. Unfortunately, due to various idiosyncrasies of hardware and software, there are several situations in which warning EVRs do not denote real anomalies (are false positives). As a result, one of the roles of the ground operations team is to label those received warnings that are to be ignored; this work needs to be completed before the next plan can be uplinked to the spacecraft. To speed up analysis, we have implemented a set of rules that can label EVRs corresponding to known idiosyncrasies. As a result, ground operators can limit their attention to only unlabeled warning EVRs. We describe some of these rules below.

The first pair of rules capture a known (benign) race condition in the software caused when a thread servicing the radio is starved and generates the warning TLM_TR_ERROR which indicates missing telemetry. This happens because the thread is preempted by higher-priority threads that are processing one of two commands (either MOB_PRM or ARM_PRM) that generate reports of current mobility and robotic arm parameter values. Because the error was discovered late in the mission, and the impact is benign, no code fix was deemed necessary. The rule below looks for this known scenario by checking for an occurrence of TLM_TR_ERROR during execution of either a MOB_PRM or an ARM_PRM command. A command execution interval itself is defined by a pair of CMD_DISPATCH and CMD_COMPLETE events whose maps agree on the cmd key, which denotes the command name.

```
cmdExec :– CMD_DISPATCH before CMD_COMPLETE
  where CMD_DISPATCH.cmd = CMD_COMPLETE.cmd
  map {cmd → CMD_DISPATCH.cmd}

okRace :– TLM_TR_ERROR during cmdExec
  where cmdExec.cmd = "MOB_PRM" | cmdExec.cmd = "ARM_PRM"
```

The next rule involves a timing consideration. In this case, an instrument power-on command fails and then recovers within 15 seconds. Since the behavior is predictable, and benign, the two warnings about command failure and subsequent recovery are labeled as being expected. The **this** keyword serves as a label for the interval that is generated.

```
okCmdFail :–  INST_PWR_ON before
              INST_CMD_FAIL before
              INST_RECOVER
  where this.end – this.begin ≤ 15
```

The last set of rules label a situation in which a warning about task starvation is expected whenever an activity (labeled vdp) which fetches data products from the cameras overlaps with an Earth communication activity (labeled comm, and identified by an id field in its map). In this case, we use the slice operator to identify the interval of overlap between the vdp and comm intervals:

```
comm :− COMM_BEGIN before COMM_END
  map {id → COMM_BEGIN.id}

vdp :− VDP_START before VDP_STOP

okStarvation :− TASK_STARVATION during (vdp slice comm)
  map {id → comm.id}
```

2.7 Performance Considerations

We saw in Section 2.4 that the complexity of the basic `nfer` processing algorithm with an idempotent selection function is $O(n^3)$ with respect to the length of the trace. For many cases, this is too high to be practical. The LogFire and Prolog experiments referred to in Section 2.1 illustrate this. Introducing a selection function to only keep intervals which are *minimal* reduces the complexity considerably, according to experiments, pushing the algorithm into the realm of being practical. However, performance is still insufficient for some cases. Kernel trace utilities such as *tracelogger* for QNX or *ftrace* for Linux can produce millions of messages per minute, for example [173, 213]. Even with the minimality selection function, `Nfer` is unable to cope with such data rates over long periods.

In this section we look for modifications which can improve the performance of the basic algorithm, anticipating, however, that such improvements may suffer from lack of soundness and completeness compared to the basic algorithm. These modifications are shown as alterations to Algorithm 1. Three such modifications are given: one-use, most-recent, and rolling-window. Note that, although they are given as modifications to the algorithm, their relationship to the semantics of `nfer` may be understood as changes to the selection function. Each of these modifications can be understood as a method to reduce the worst-case cardinality of the data structures over which Algorithm 1 iterates, specifically limiting the size of the left and right caches.

2.7.1 Proposed Modifications

The **one-use** modification, shown as Algorithm 2 (changes are underlined), deletes input intervals when they are used to create new intervals. In this way, any interval may only be used to create one new interval per rule. The *New* variable is modified to hold triples

$(i_{\text{left}}, i_{\text{right}}, i_{\text{new}})$ for each new candidate interval i_{new} included (before selection), where i_{left} and i_{right} are the intervals in the left and right caches, respectively, that i_{new} is derived from. This is needed to remove those intervals from the caches later if i_{new} is selected (lines 25 and 26). The one-use modification reduces the maximum sizes of the left and right caches. This also reduces the *amortized* worst-case cardinality of the *New* set to one. Note that, because the worst-case cardinality of the *Produced* set is still linear in the size of the trace, the complexity of the minimality selection function is also linear, so the worst-case complexity of the `nfer` processing function is not changed.

Algorithm 2 One-use Modification

```
 1: ···
 4: if interval.name = rule.leftLabel then
 5:     ···
 9:     if ¬ exclude then
10:         New ← New ∪ $\underline{\{(\text{interval}, \varepsilon, \text{rule.createInterval(interval)})\}}$
11: else
12:     for rightIntv ∈ rule.RightCache do
13:         if rule.testInclusion(interval, rightIntv) then
14:             New ← New ∪ $\underline{\{(\text{interval}, \text{rightIntv}, \text{rule.createInterval(interval, rightIntv)})\}}$
15: if interval.name = rule.rightLabel ∧ rule is *inclusive* then
16:     for leftIntv ∈ rule.LeftCache do
17:         if rule.testInclusion(leftIntv, interval) then
18:             New ← New ∪ $\underline{\{(\text{leftIntv}, \text{interval}, \text{rule.createInterval(leftIntv, interval)})\}}$
19: ···
23: for $\underline{\text{(left, right, new)}}$ ∈ select(New, rule.Produced) do
24:     rule.Produced ← rule.Produced ∪ {new}
25:     $\underline{\text{rule.LeftCache} \leftarrow \text{rule.LeftCache} \setminus \{\text{left}\}}$
26:     $\underline{\text{rule.RightCache} \leftarrow \text{rule.RightCache} \setminus \{\text{right}\}}$
27:     process(new)
```

The **most-recent** modification, shown as Algorithm 3, only stores the most recent intervals instead of keeping a cache of all of the previously seen ones. This change reduces the maximal cardinality of all caches to 1, except for the *Produced* cache. A cache's single element can be selected with the *head* function. The variable *New* holds at most one interval in this solution. Because the *Produced* cache still has a worst-case cardinality of $n-1$ (suppose a rule A :− B **before** B, then an interval is created for each subsequent B after the first), the worst-case complexity of the `nfer` processing algorithm is not reduced.

Algorithm 3 Most-recent Modification

```
18: ···
19: if interval.name = rule.leftLabel then
20:     rule.LeftCache ← $\underline{\{interval\}}$
21: if interval.name = rule.rightLabel then
22:     rule.RightCache ← $\underline{\{interval\}}$
```

The **rolling-window** modification, shown as Algorithm 4, only considers intervals in a cache that occur within a time window. Intervals falling outside the window are deleted from the caches. The time window is calculated as a fixed offset from the end of the last submitted interval. The rolling-window modification does not change the maximum cardinality of the caches (if all events occur within the window, then nothing is deleted), so it does not change the complexity of the algorithm according to the formula. However, if the window size is carefully chosen, the heuristic can have a drastic effect in the execution time of the processing algorithm.

2.7.2 Experimental Setup

We conducted a series of screening experiments to explore possibilities for improving the execution time of the `nfer` processing algorithm. We do not intend for this to be a comprehensive evaluation for choosing one algorithm over another. We implemented each algorithm and used it to apply `nfer` specifications on three different test datasets. All algorithms were tested using the *minimality* selection function discussed in Section 2.4.3. Both the C and Scala implementations were used to run the experiments. This helped us to eliminate some implementation specific blocking factors that could affect the performance of an algorithm. The C implementation experiments were performed in the Linux 4.9.6 operating system on an Intel Core i5 running at 2.4 GHz with 16 GB of RAM. The Scala implementation experiments were performed in the Mac OS X 10.10.5 operating system on an Intel Core i7 running at 2.8 GHz with 16 GB of RAM. The datasets are described in the following paragraphs.

Sequential Sense-Process-Send (SSPS)

The Sequential Sense-Process-Send (SSPS) dataset was generated by a system mimicking an embedded data collection device. The device-under-test (DUT) was a first generation

Algorithm 4 Rolling-window Modification

```
 6: ···
 7: for rightIntv $\in$ rule.RightCache do
        if $\underline{\text{rightIntv.end} < \text{interval.end - WINDOW}}$ then
            $\underline{\text{rule.RightCache} \leftarrow \text{rule.RightCache} \setminus \{\text{rightIntv}\}}$
        else
 8:         exclude $\leftarrow$ exclude $\vee$ rule.testExclusion(interval, rightIntv)
 9: ···
12: for rightIntv $\in$ rule.RightCache do
        if $\underline{\text{rightIntv.end} < \text{interval.end - WINDOW}}$ then
            $\underline{\text{rule.RightCache} \leftarrow \text{rule.RightCache} \setminus \{\text{rightIntv}\}}$
        else
13:         if rule.testInclusion(interval, rightIntv) then
14:             New $\leftarrow$ New $\cup$ {rule.createInterval(interval, rightIntv)}
15: ···
16: for leftIntv $\in$ rule.LeftCache do
        if $\underline{\text{leftIntv.end} < \text{interval.end - WINDOW}}$ then
            $\underline{\text{rule.LeftCache} \leftarrow \text{rule.LeftCache} \setminus \{\text{leftIntv}\}}$
        else
17:         if rule.testInclusion(leftIntv, interval) then
18:             New $\leftarrow$ New $\cup$ {rule.createInterval(leftIntv, interval)}
19: ···
23: for new $\in$ select(New, $\{p : p \in$ rule.Produced, $p$.end $>$ interval.end - WINDOW$\}$ )
    do
24:     rule.Produced $\leftarrow$ rule.Produced $\cup$ {new}
25:     process(new)
```

BeagleBone with a 720 MHz ARM Cortex-A8 running version 6.6.0 of the QNX real-time operating system [188]. Logs were collected using the QNX tracelogger utility. The SSPS system executes a periodic process with three phases: acquisition, processing, and communication. Data acquisition is accomplished using the *dd* utility to copy random data to a file. There are two phases of data processing. The first data processing phase creates a checksum for the data using *cksum* and the second compresses the data using *bzip2*. The communication phase is optional, but if it is performed it consists of using *scp* to transfer the compressed file to another host on the network. After the optional communication phase, the process is made periodic by sleeping for a period.

The system logs an event between each phase, and between the two parts of the processing phase. This complicates log comprehension, as each event must serve as both the end of the previous phase and the beginning of the next. The `nfer` specification for the SSPS dataset, shown in Figure 2.10, defines each phase, then groups the phases into `HANDLING` and `FINALIZATION` intervals, and finally defines a `MAIN_LOOP` interval using those. `HANDLING` is made up of the acquisition and processing phases, while finalization is made up of communication and sleep. Since the communication phase and the event associated with its beginning are optional, the rules must allow for a processing phase that ends with either communication or sleep.

The tested dataset contained 12,766 relevant events covering a collection period of approximately 26 hours. The SSPS dataset is also described in Section 2.8.5.

```
ACQUISITION <− EV401 before EV402
PROC1 <− EV402 before EV403
PROC2_W_COM <− EV403 before EV404
PROC2_NO_COM <− (EV403 before EV405) unless contain EV404
COMMUNICATION <− EV404 before EV405
SLEEP <− EV405 before EV401
PROCESSING <− PROC1 meet PROC2_W_COM
PROCESSING <− PROC1 meet PROC2_NO_COM
HANDLING <− ACQUISITION meet PROCESSING
FINALIZATION <− COMMUNICATION meet SLEEP
FINALIZATION <− SLEEP unless follow COMMUNICATION
MAIN_LOOP <− HANDLING meet FINALIZATION
```

Figure 2.10: Specification used in the SSPS experiments

System Call Logs with Natural Random Faults (LANL)

The System Call Logs with Natural Random Faults (LANL) dataset was generated by running a simulation of an automotive cruise-control application on a computer under high-energy neutron bombardment [173]. The DUT was a Xilinx ZC706 featuring a XC7Z045 [233] System-on-a-Chip (SoC) running version 6.6.0 of the QNX real-time operating system [188]. Faults in the dataset were generated by placing the SoC in the path of a high-energy neutron beam at the Los Alamos Neutron Science Center (LANSCE) facility at the Los Alamos National Laboratory (LANL) in New Mexico, USA. The cruise-control application used for the LANL dataset consists of communicating components: a speed sensor, brake and accelerator actuators, and cruise controllers. The speed sensor is simulated and sends randomized speed readings to a configurable number of cruise controller functions which then command the brake or accelerator actuators to correct for any changes. Multiple cruise controller functions operate in parallel and their actuator commands are compared against one another as a form of error correction. The ionizing radiation present during system operation can cause differences in the calculated actuator commands which result in changes to the trace.

The `nfer` specification for the LANL dataset, shown in Figure 2.11, defines periods of activity for each component and then defines the nominal and off-nominal behavior during those periods. During the period of activity for the speed sensor, the nominal behavior is for the speed value to be sent and no off-nominal behavior is defined. During the period of activity for the actuators, the nominal behavior is for a unanimous response to be received after a request to the controllers for commands is sent and the off-nominal behavior is for a non-unanimous or non-quorum response to be received. During the period of activity for the controllers, the nominal behavior is for the correct actuator to acknowledge receipt after a controller sends its command and the off-nominal behavior is for an incorrect actuator to acknowledge receipt of the command.

The tested dataset contained 50,000 relevant events covering a collection period of approximately 10 hours. The LANL dataset is also described in Section 2.8.5.

Mars Science Laboratory (MSL)

The Mars Science Laboratory (MSL) dataset was generated by checking the property okRace described in Section 2.6 on telemetry logs received from the Curiosity rover. We checked logs covering rover activities over around 60 days. For convenience, we first filtered the rover logs to include only relevant EVRs. These EVRs are generated on board the

```
SENSOR <- EV20 before EV21
ACTUATOR <- EV22 before EV23
CONTROLLER <- EV24 before EV25
SENSOR_OK <- EV12 during SENSOR
BRAKE_PRESSURE_OK <- EV14 before before EV1
ACCEL_PRESSURE_OK <- EV14 before EV4
ACTUAT_BRAKE_OK <- BRAKE_PRESSURE_OK during ACTUATOR
ACTUAT_ACCEL_OK <- ACCEL_PRESSURE_OK during ACTUATOR
ACTUATOR_OK <- ACTUAT_BRAKE_OK coincide ACTUAT_ACCEL_OK
RESP_ACTUAT <- EV9 before EV10
RESP_SENSOR <- EV8 before EV10
CTRL_ACTUAT_OK <- RESP_ACTUAT during CONTROLLER
CTRL_SENSOR_OK <- RESP_SENSOR during CONTROLLER
CONTROLLER_OK <- CTRL_ACTUAT_OK unless contain RESP_SENSOR
CONTROLLER_OK <- CTRL_SENSOR_OK unless contain RESP_ACTUAT
BRAKE_PRESSURE_WARN <- EV14 before before EV2
ACCEL_PRESSURE_WARN <- EV14 before EV5
ACTUATOR_WARN <- BRAKE_PRESSURE_WARN during ACTUATOR
ACTUATOR_WARN <- ACCEL_PRESSURE_WARN during ACTUATOR
BRAKE_PRESSURE_ERROR <- EV14 before before EV3
ACCEL_PRESSURE_ERROR <- EV14 before EV6
ACTUATOR_ERROR <- BRAKE_PRESSURE_ERROR during ACTUATOR
ACTUATOR_ERROR <- ACCEL_PRESSURE_ERROR during ACTUATOR
ACTUATOR_ERROR <- EV7 during ACTUATOR
```

Figure 2.11: Specification used in the LANL experiments

rover when the software executes command sequences as part of daily activity plans that are uplinked to the rover from Earth. The test dataset contained 50,000 relevant EVRs.

2.7.3 Experimental Results

Table 2.3 shows the results from the experiments. The **Impl** column differentiates between the C and Scala implementation (note that comparisons in timing and memory use should not be made between the C and Scala implementations). The **Algorithm** column shows the version of the processing algorithm under consideration, where Basic means Algorithm 1, One-use means Algorithm 2, Most-recent means Algorithm 3, and Window means Algorithm 4. The Window algorithm was applied with different window sizes indicated in seconds. The **Data** column shows the dataset and specification used for the row. The **Ex Time** column shows the clock time in seconds used by the program to reach a fixed-point, and the **Memory** column shows the peak memory used during computation. The **Precision** and **Recall** columns show the precision and recall of the algorithm where the "true" output is the result of running the Original algorithm.

Precision is an indicator for soundness (wrt. the basic algorithm) and is defined as the fraction of created intervals that are correct, and recall is an indicator for completeness (wrt. the basic algorithm) and is defined as the fraction of all expected intervals that are generated. More precisely, given the set of intervals B created by running the basic Algorithm 1 on a dataset with a specification, and given the set of new intervals N created by running a different algorithm on the same dataset and specification, precision and recall are defined as:

$$\text{precision} \ = \frac{|B \cap N|}{|N|} \qquad \text{recall} \ = \frac{|B \cap N|}{|B|}$$

For example, Line 4 of Table 2.3 shows the results from running the C implementation of the rolling-window algorithm (with a window size of five seconds) on the SSPS dataset with its `nfer` specification. It took 0.01 seconds to complete and used 11 megabytes of memory at its peak. The number of intervals found by the basic algorithm for the dataset and specification was 19,360, the number of intervals found by the rolling-window modification (5 s) was 5,749, and the size of the intersection between the two results was 5,661. So, the precision was $5661/5749 = 0.985$ and the recall was $5661/19360 = 0.292$.

In case of the MSL dataset, the recall number is followed by a number in parentheses. This is the number of `okRace` intervals produced, which are the intervals in which we are ultimately interested for this dataset. Four such intervals should be produced as shown for the Basic algorithm. We observe that the Most-recent algorithm did not produce any of these, and that the Window algorithm with a window of size 30 seconds (the smallest shown) only produced two of these, whereas the remaining window sizes produced all four.

Table 2.3 shows evidence that the rolling-window modification has the most promise to improve performance while still finding most or all of the relevant intervals. For each dataset, a window size was found that enabled a precision and recall of exactly 1.0, while executing at least an order of magnitude faster than the original algorithm. This ideal examined window size was different for each dataset: it was 210 seconds for SSPS, 1,000 seconds for LANL, and 41,000 seconds for MSL. For each new application, the window size must be tuned to find this optimal number.

The one-use modification was interesting, in that it was able to return nearly the same results as the basic algorithm but it did not show as much of a performance improvement as the well-tuned rolling-window modification. For the LANL dataset, the results were *exactly* the same as the basic algorithm. For the SSPS dataset 25 out of 19,360 intervals were missing and three extra intervals were generated. For the MSL dataset around 9% of the expected results were missing and one extra interval was generated. Performance varied from about a 32% improvement for MSL to about a 80% improvement for LANL, but this is in contrast to the one or two order of magnitude improvements seen from the other methods.

The most-recent modification missed too many relevant intervals to be of much practical use, although it was found to execute quickly and had good precision. Its recall of only 0.613 for the LANL dataset and 0.454 for the MSL dataset show that it found too few of the expected results. The modification had nearly perfect results for the SSPS dataset, however, so it may be usable in some circumstances.

2.8 Mining Intervals from Real-Time System Traces

To use the results from `nfer` to improve telemetry comprehension, rules must specify the interval abstractions to infer from a trace of events. Typically, such a specification would be written by the engineers that designed the software. Writing specifications is time-consuming and error-prone, however, and only captures facets of the software that were understood at design time. Dynamic specification mining seeks to solve these problems by using machine learning techniques to discover rules that define how the system should behave.

We can leverage prior research in pattern recognition to mine ATL relations from historical telemetry, but these works assume the existence of a multivariate interval series.

Table 2.3: Results from experiments

Impl	**Algorithm**	**Data**	**Ex Time**	**Memory**	**Precision**	**Recall**
C	Basic	SSPS	1.58 s	16 MB	1.0	1.0
C	One-use	SSPS	0.37 s	13 MB	$\sim$ 1.0	0.999
C	Most-recent	SSPS	0.14 s	9 MB	$\sim$ 1.0	0.997
C	Window (5 s)	SSPS	0.01 s	11 MB	0.985	0.292
C	Window (15 s)	SSPS	0.01 s	13 MB	0.999	0.699
C	Window (30 s)	SSPS	0.02 s	15 MB	$\sim$ 1.0	0.966
C	Window (60 s)	SSPS	0.02 s	15 MB	$\sim$ 1.0	$\sim$ 1.0
C	Window (210 s)	SSPS	0.03 s	15 MB	1.0	1.0
C	Window (1,000 s)	SSPS	0.06 s	15 MB	1.0	1.0
C	Basic	LANL	138.40 s	54 MB	1.0	1.0
C	One-use	LANL	26.93 s	42 MB	1.0	1.0
C	Most-recent	LANL	0.55 s	12 MB	1.0	0.613
C	Window (1 s)	LANL	0.18 s	49 MB	1.0	0.772
C	Window (5 s)	LANL	0.21 s	53 MB	1.0	0.986
C	Window (30 s)	LANL	0.30 s	52 MB	1.0	0.999
C	Window (500 s)	LANL	1.96 s	54 MB	1.0	$\sim$ 1.0
C	Window (1,000 s)	LANL	3.63 s	53 MB	1.0	1.0
C	Window (10,000 s)	LANL	41.59 s	53 MB	1.0	1.0
Scala	Basic	MSL	251.1 s	80 MB	1.0	1.0 (4)
Scala	One-use	MSL	196.9 s	70 MB	$\sim$ 1.0	0.916 (4)
Scala	Most-recent	MSL	0.4 s	1 MB	0.997	0.454 (0)
Scala	Window (30 s)	MSL	2.9 s	1 MB	$\sim$ 1.0	0.550 (2)
Scala	Window (100 s)	MSL	27.3 s	10 MB	0.976	0.913 (4)
Scala	Window (2,000 s)	MSL	24.7 s	50 MB	0.992	0.972 (4)
Scala	Window (41,000 s)	MSL	28.5 s	40 MB	1.0	1.0 (4)
Scala	Window (500,000 s)	MSL	89.6 s	70 MB	1.0	1.0 (4)
Scala	Window (1,000,000 s)	MSL	166.0 s	80 MB	1.0	1.0 (4)

To use existing research in interval pattern mining, we must first convert our telemetry (sometimes called a symbolic time sequence [160]) to a multivariate interval series. To this end, we introduce an algorithm for automatically mining time intervals from real-time system traces. We show that our algorithm has linear complexity in the size of the trace and that it produces interesting intervals.

Our requirements lead to an approach that combines aspects of sequential pattern mining and automata-based specification mining. We are interested in finding intervals that have been called *response* patterns [78] or *serial episodes* [157]. We mine patterns in event sequences that define the beginning and the end of a *process* in a real-time embedded system such as a spacecraft. A process may be any task, routine, or function that is part of the behavior of the system. Since our technique outputs `nfer` rules, the mined response patterns are expressed as ATL **before** relations.

2.8.1 Problem

Given a trace $\tau \in \mathbb{T}$, our goal is to find pairs of event names $(\eta_1, \eta_2) \in \mathcal{I} \times \mathcal{I}$ such that $\exists (\eta_1, t_1), (\eta_2, t_2) \in \tau$ where $t_1 < t_2$, such that $\forall (\eta_1, t_j), (\eta_2, t_k) \in \tau$ the intervals $(\,\cdot\,, t_j, t_k) \in \mathbb{I}$ are *interesting*.

It is simple to convert an event trace into a sequence of atomic intervals, but this is insufficient for our purposes. Any event may be converted to an atomic interval in the same fashion as the initial pool is generated in Section 2.3. However, atomic intervals are not more useful than the original events for deriving meaning from the trace or as the input to an algorithm meant to mine ATL relations.

We observe that an algorithm to mine Allen's or other temporal relations on an interval series will not work on an atomic interval sequence. The only Allen relation that can be applied to an atomic interval sequence to define intervals is **before**. We designed our algorithm therefore to mine **before** relations from a trace of events. The intervals that result from the mined relations may then be used for human or machine comprehension, or combined with traditional algorithms meant to mine temporal relations from an interval series.

2.8.2 Measuring Interestingness

We must also address the problem of whether or not intervals are *interesting*. Although the concept is subjective, we can observe properties that make some intervals more interesting than others. It is often assumed that an interesting interval is one that is derived from a rule that also appears in a handwritten specification. This definition is only helpful for judging the quality of a mining algorithm, however. We must define metrics that we can use to mine intervals when no specification exists.

Minimality

The example in Figure 2.12 illustrates why minimality is a property of interesting intervals. The intervals 1, 2, and 3 all match the relation A **before** B, but only 1 and 2 are minimal. If A is the beginning of a process and B is its end, then interval 3 does not capture the intent of the relation.

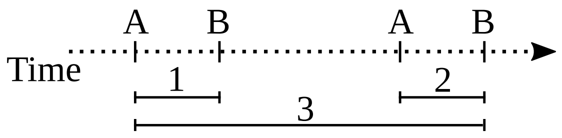


Figure 2.12: Minimality example

Support and Confidence

We use the two most common metrics used to judge the acceptance of rules in specification or pattern mining: *support* and *confidence*. Support is usually defined as the number of times a rule is matched in the training data, and confidence is the probability that the post-condition of the rule follows the pre-condition of the rule.

Some research has suggested that these common metrics are less effective than other statistical measurements for judging the correctness of a rule. Le and Lo published a study that compared the effectiveness of different metrics in mining response and precedence patterns from the Java Software Development Kit (SDK) source code [144]. In their work, they found that the "odds ratio" and "leverage" metrics outperformed support and confidence on average. Their definitions for computing the leverage and odds ratio metrics for a relation A **before** B include the notion of tracking the probability that A occurs independently and tracking the probability that neither A nor B occur.

However, Le's research makes assumptions that do not apply to this work. Most importantly, they assume the use of a sliding-window algorithm which is not sufficient to mine intervals of arbitrary duration. In our algorithm, which does not use a sliding window, the probability of A occurring independently will be either one or zero, and the same applies to the probability of neither A nor B occurring. Additionally, they used instrumented Java code in their research, where they added logging to the beginning of each method. Our traces represent telemetry or similar system traces logged from real-time systems, where the data is periodic, and events often occur at the *end* as well as the beginning of processes.

Relative Duration

We make the assumption that the average duration of an interval that represents the execution of a process will usually be shorter than the average duration of the time between executions. The real-time software that generates the event streams from which we mine intervals is made up of processes that are mostly *periodic*. Even when they can be interrupted, tasks are usually executed in a cycle where the system wakes up, performs work, and shuts down.

In developing our algorithm, the periodic nature of embedded software presented a challenge. In most cases, when the support and confidence thresholds are met for a relation A **before** B, they are also met for the relation B **before** A. In a periodic process, one of those intervals represents the time between when a process starts and ends (its duration) and the other represents the time between when it ends and starts again. The time between when a process ends and when it starts again is called its between-job inter-arrival time (BJI). If a process is interrupted its duration may be extended. The time during which it is interrupted is called its intra-job inter-arrival time (IJI).

Figure 2.13 shows a trace and the intervals that represent a periodic task execution. In the figure, A marks the beginning of the task and B marks the end, while I marks the beginning of an interrupt service routine (ISR) and J marks its end. Interval 1 shows an IJI of the task, interval 3 shows a BJI of the task, and intervals 2 and 4 show instances of the task's duration.

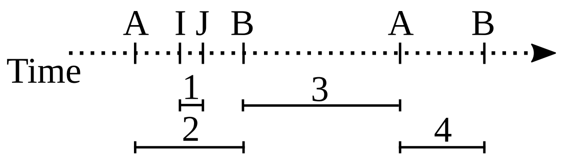


Figure 2.13: Periodic processes

Without further heuristics, it is not possible to differentiate between the BJI and duration of a periodic process. When a process repeats, the result will be an alternating pattern of A and B events. Since we do not assume a complete trace, we cannot simply treat the first of the two to appear in the trace as the beginning of the process' execution.

A recent work by Iegorov et al. used the relationship between the IJI and the BJI of a process to mine strictly periodic tasks and their response times [118]. They used an assumption that the IJI of a process should be shorter than its BJI on average. This assumption comes from the predictable way that real-time software is scheduled and the fact that such systems are designed with a maximum expected CPU utilization. We use the same idea to differentiate between the duration of a process and its BJI.

We performed a simulation using the YAO-SIM tool [37] to see if a relationship exists between the duration of a task and its BJI that could be used to differentiate between them. Following a similar procedure to [118] and [37], we generated 1,000 periodic task sets for CPU utilizations between 10% and 70% at 5% increments and simulated execution of the system for 1×10^7 time units. We used two scheduling algorithms common in real-time embedded systems: Rate Monotonic (RM) scheduling with fixed priorities, and Earliest Deadline First (EDF) scheduling with dynamic priorities. The number of tasks in each task set and their Worst-Case Execution Times (WCETs) were randomly chosen from a uniform distribution and the period of each task was computed from its WCET and the CPU utilization using the UUniFast algorithm [38]. The number of tasks was selected from the interval $[3, 10]$ and the WCET was taken from the interval $[1, 30]$.

Figure 2.14 shows the results of a scheduling simulation comparing the duration of a task to its BJI. The y-axis represents the ratio of the mean of duration for a task set over the mean of BJI, while the x-axis represents the CPU utilization of the system. At each CPU utilization level, the results using EDF scheduling are on the left and those using RM scheduling are on the right. Dots that appear below the horizontal line represent simulations where a task's average duration was shorter than its BJI, while dots that appear above the line represent simulations where the reverse was true.

The choice of 70% CPU utilization as the upper bound of the simulations was deliberate. The commonly accepted notions of safe CPU utilization in real-time systems are that 69% is the theoretical safe upper limit, 51-68% is considered "safe", 26-50% is considered "very safe", and below 26% is considered "unnecessarily safe" [143]. These regions are shown in Figure 2.14 by differently shaded backgrounds. For any system in the "very safe" or "unnecessarily safe" regions, our simulations show that any task's duration should always be shorter than its BJI. For systems in the "safe" region, this assumption still holds most of the time.

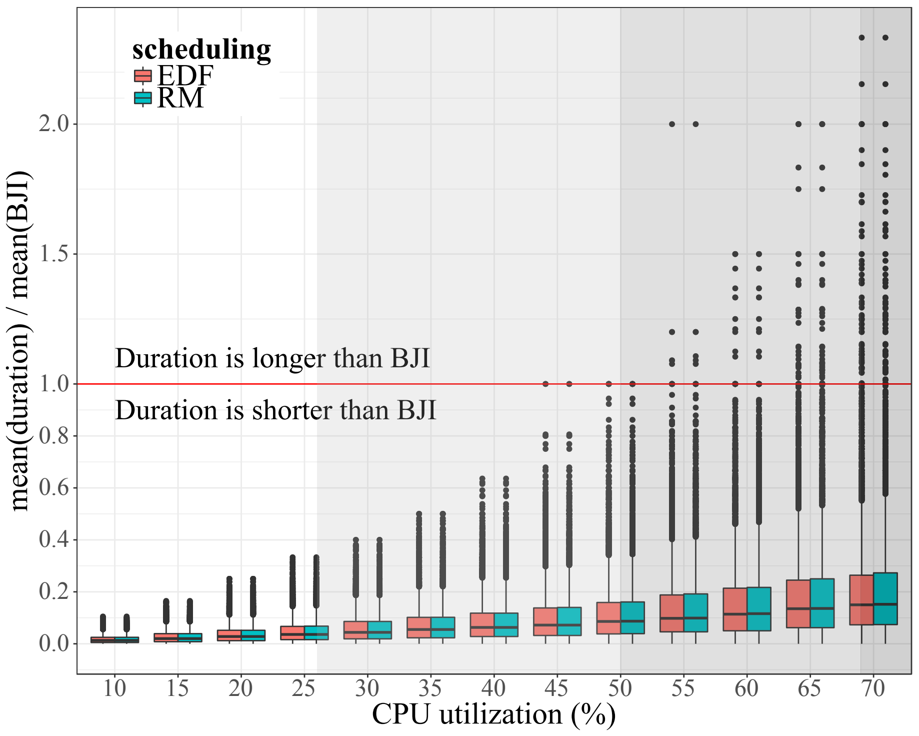


Figure 2.14: Simulation results comparing duration with BJI

2.8.3 Mining Algorithm

This section describes our solution to mine pairs of event names which correspond to the beginnings and ends of minimal intervals. The details are shown in Algorithm 5. Our algorithm is loosely based on work by Cutulenco et al. to mine Timed Regular Expressions (TREs) [57] in that we achieve linear asymptotic complexity in the length of the trace using a matrix of pattern statistics.

The idea behind the algorithm is as follows. We process the trace in order. When we encounter an event with name A, we store it as the most recent A event. For every succeeding event B, we count the relation A **before** B as *matched* and record the duration of the interval. When the next A event is encountered, we increment a success counter for all A **before** $\cdot$ relations marked as matched exactly once and increment a failure counter otherwise. We also update the average duration of successful relations. When all events have been read, we check each pair of event names and output the associated rule if the user specified confidence and support thresholds are exceeded and if the average duration is shorter than that of the inverse relation.

We create a square matrix data structure $\mathcal{M}$ where each dimension maps to an event name: a member of $\mathcal{I}$. The rows of $\mathcal{M}$ represent the left side of a **before** relation, and the columns represent the right side. Each entry $\mathcal{M}(i, j)$ in the matrix contains a 5-tuple (m, s, f, pd, ad) where $m \in \mathbb{N}$ represents the number of times the event name $\mathcal{I}(j)$ has been seen since the last $\mathcal{I}(i)$ (the *matched* count), $s, f \in \mathbb{N}$ are *success* and *failure* counts, and $pd, ad \in \mathbb{R}$ are the *previous duration* and the *average duration* of successfully matched pairs. We also keep an array $\mathcal{R}$, where indices map to names in $\mathcal{I}$ and contain the most recent copy of each event if one has been seen.

The user must specify a support threshold $\mathcal{S}_t$ and a *confidence* threshold $\mathcal{C}_t$. A count is kept of the *success* and *failure* of each event to imply every other event. The support for a pair is defined as the total number of successes for that pair. This varies somewhat from other notions of support because we do not use a sliding window in our algorithm and events may occur many times in the trace. The confidence for a pair is defined as the number of successes for that pair over the sum of the successes and failures.

After the trace is complete, the matrix is iterated over, and each cell is checked to see if it meets the acceptance conditions. A pair $(\mathcal{I}(i), \mathcal{I}(j))$ is accepted if the following conditions hold.

1. The event is not matching itself: $i \neq j$.
2. The confidence threshold is met: $\frac{\mathcal{M}(i,j).s}{\mathcal{M}(i,j).s+\mathcal{M}(i,j).f} \geqslant \mathcal{C}_t$.
3. The support threshold is met: $\mathcal{M}(i, j).s \geqslant \mathcal{S}_t$.
4. The average duration is less than the average duration of the inverse pair: $\mathcal{M}(i, j).ad < \mathcal{M}(j, i).ad$.

After a trace has been processed, the matrix $\mathcal{M}$ is traversed in a finalization step that facilitates handling multiple, non-contiguous traces. For each cell of the matrix, its *success* count is incremented if its *matched* count is exactly one and its *failure* count is incremented if its *matched* count is *greater than* one. Importantly, its *failure* count is not incremented if the *matched* count is zero. This causes the algorithm to assume that the behavior after the trace will not invalidate any of the relations. The matched count of each cell is also set to zero and the array of the most recent events ($\mathcal{R}$) is cleared. This final step enables handling traces with few events such as the example in Section 2.8.4 and the case study in Section 2.8.5.

Algorithm 5 Interval Mining Algorithm

```
procedure ADDTOMODEL($event$)
    define $i : \mathcal{I}(i) = event.name$
    for $left \in \mathcal{I}.indices$ do
        if $\mathcal{R}(left)$ is set then
            if $\mathcal{R}(left).time < event.time$ then        ▷ if the before relation holds
                increment $\mathcal{M}(left,\ i).matched$
                $\mathcal{M}(left,\ i).pd \leftarrow event.time - \mathcal{R}(left).time$
    end for
    if $\mathcal{R}(event.name)$ is set then
        for $right \in \mathcal{I}.indices$ do
            if $\mathcal{M}(i,\ right).matched = 1$ then
                increment $\mathcal{M}(i,\ right).success$
                $toadd \leftarrow (\mathcal{M}(i,\ right).pd - \mathcal{M}(i,\ right).ad)/\mathcal{M}(i,\ right).success$
                $\mathcal{M}(i,\ right).ad \leftarrow \mathcal{M}(i,\ right).ad + toadd$
            else
                increment $\mathcal{M}(i,\ right).failure$
            $\mathcal{M}(i,\ right).matched \leftarrow 0$
        end for
    $\mathcal{R}(event.name) \leftarrow event$
```

2.8.4 Mining Example

In this section, we present an illustrative example of the mining algorithm discussed in Section 2.8.3. Assume a confidence threshold $\mathcal{C}_t$ of 1.0, a support threshold $\mathcal{S}_t$ of 1, and the trace shown in Figure 1.2: ⟨ (A, 10), (B, 20), (B, 30), (C, 40), (A, 50), (C, 60) ⟩. The state of the matrix $\mathcal{M}$ and the array $\mathcal{R}$ are shown after each event is added to the model. Each field of $\mathcal{M}$ contains the 5-tuple (m, s, f, pd, ad), and each field of $\mathcal{R}$ contains the most recent event with that name.

- ADDTOMODEL(**(A,10)**) – The indices corresponding to A, B, and C in $\mathcal{I}$ are looped over on Line 3, but $\mathcal{R}$ is empty so the condition on Line 4 is false and the procedure continues on Line 9. The condition on Line 9 is also false because $\mathcal{R}$ is empty, so the event is simply inserted into $\mathcal{R}$ on Line 19.

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
B	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,10)
B	
C	

- ADDTOMODEL(**(B,20)**) – There is now an A event in $\mathcal{R}$, so the conditional is taken for that event on Line 4 and (A,10) is compared to (B,20) on Line 5. Since $10 < 20$, *matched* is incremented for the pair (A,B) on Line 6 and its *previous duration* is set on Line 7. No B event is in $\mathcal{R}$, so the condition on Line 9 is false. The event is inserted into $\mathcal{R}$ on Line 19.

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,0,0,0,0	1,0,0,10,0	0,0,0,0,0
B	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,10)
B	(B,20)
C	

- ADDTOMODEL(**(B,30)**) – There are now A and B events in $\mathcal{R}$, so the conditional on Line 4 is taken for those. The matched count for (A,B) and (B,B) are incremented, and their *previous durations* are set. The pair (A,B) has been seen twice, so its matched count is now two. For the first time, the condition on Line 9 is true, since B has been seen before. The indices corresponding to A, B, and C in $\mathcal{I}$ are looped over on Line 10 and the pairs (B,A), (B,B), and (B,C) are checked to see if matched equals one on Line 11. This is only true for (B,B), so its *success* counter is incremented on Line 12 and its *average duration* is updated on Line 14. The failure counters for (B,A) and (B,C) are incremented on Line 16. All three *matched* counters are reset to zero on Line 17. Finally, the index for B in $\mathcal{R}$ is replaced with (B,30).

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,0,0,0,0	2,0,0,20,0	0,0,0,0,0
B	0,0,1,0,0	0,1,0,10,10	0,0,1,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,10)
B	(B,30)
C	

- addToModel(**(C,40)**) – There are both A and B events in $\mathcal{R}$, so the conditional on Line 4 is taken for those. Both (A,C) and (B,C) have their *matched* counters incremented, their *previous durations* are recorded, and the event is inserted into $\mathcal{R}$.

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,0,0,0,0	2,0,0,20,0	1,0,0,30,0
B	0,0,1,0,0	0,1,0,10,10	1,0,1,10,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,10)
B	(B,30)
C	(C,40)

- addToModel(**(A,50)**) – All three event names have been seen, so the condition on Line 4 is true in all cases. The pairs (A,A), (B,A), and (C,A) all have their *matched* counters incremented on Line 6 and their *previous durations* updated on Line 7. The condition on Line 9 is true again, so the pairs (A,A), (A,B), and (A,C) are checked to see if they have been matched once. The pairs (A,A) and (A,C) have been matched once, so their *success* counters are incremented and average durations updated. However, (A,B) has been matched twice, so its *failure* counter is updated. All three *matched* counters are reset. The index for A in $\mathcal{R}$ is replaced with (A,50).

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,1,0,40,40	0,0,1,20,0	0,1,0,30,30
B	1,0,1,20,0	0,1,0,10,10	1,0,1,10,0
C	1,0,0,10,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,50)
B	(B,30)
C	(C,40)

- ADDTOMODEL(**(C,60)**) – The pairs (A,C), (B,C), and (C,C) have their *matched* counters incremented and *previous durations* updated. On Line 12, the *success* counter for the pairs (C,A) and (C,C) are incremented and their *average duration* updated, as they have been matched, while on Line 16 the *failure* counter for the pair (C,B) is incremented since its *matched* count is zero. The event is added to $\mathcal{R}$, and the trace is complete.

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,1,0,40,40	0,0,1,20,0	1,1,0,10,30
B	1,0,1,20,0	0,1,0,10,10	2,0,1,30,0
C	0,1,0,10,10	0,0,1,0,0	0,1,0,20,20

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	(A,50)
B	(B,30)
C	(C,60)

- Finalize – After the trace has completed, each cell of the matrix is iterated over, and the *success* count is incremented if the cell's *matched* count is exactly one. The *success* counters are incremented for (B,A) and (A,C) and their *average durations* are updated. Failure counts are only incremented if the call's *matched* count is greater than one, so this is done for (B,C). All *matched* counts are reset to zero and $\mathcal{R}$ is cleared.

$\mathcal{M}$:

$\mathcal{I}$	A	B	C
A	0,1,0,40,40	0,0,1,20,0	0,2,0,10,20
B	0,1,1,20,20	0,1,0,10,10	0,0,2,30,0
C	0,1,0,10,10	0,0,1,0,0	0,1,0,20,20

$\mathcal{R}$:

$\mathcal{I}$	Most Recent
A	
B	
C	

The result of processing the example trace is that the pair (C,A) is output. Although (C,A) only appears once in the *visible* trace, the trace is assumed to be incomplete and contains no evidence to refute the hypothesis that A **before** C is an interesting relation.

The other pairs in $\mathcal{M}$ are rejected for the following reasons.

- (A,A), (B,B), (C,C) – fail condition 1, that the event is not matching itself
- (B,A), (A,B), (C,B), (B,C) – fail condition 2, that the confidence threshold of 1.0 is met

- (A,B), (C,B), (B,C) – fail condition 3, that the support threshold of 1 is met
- (A,C) – fails condition 4, that the average duration is shorter than that of the inverse pair

2.8.5 Mining Case Studies

We performed a series of case studies to demonstrate the viability of our mining algorithm. We ran the algorithm on datasets for which we had a handwritten `nfer` specification and compared the rules in those specifications to those that our algorithm mined from the same datasets. Although `nfer` rules concern a hierarchy of intervals, we were only interested in comparing against **before** relations on events rather than general ATL relations on intervals.

We implemented our algorithm in the C programming language version of `nfer`. Our tool currently loads the entire trace into memory before processing, but this is an implementation detail, not a requirement of the algorithm. We ran the command-line version of the tool, compiled using the GNU Compiler Collection (GCC) 4.9.4 with -O3 optimizations.

Experiments were performed in Linux 4.9.6 on an Intel Core i5 at 2.4 GHz with 16 GB of RAM. We obtained execution time information from the GNU *time* command and memory usage from the Valgrind Massif heap profiler. We ran each experiment 20 times and took the mean of their execution times, while the memory usage was fully deterministic.

All experiments were run with the confidence threshold $\mathcal{C}_t = 0.90$ and the support threshold $\mathcal{S}_t = 10$. Tuning these parameters is difficult without an established ground truth the algorithm is attempting to match for a dataset. We found, through experimentation, that these values were effective in cases where such a ground truth was known.

SSPS Dataset

The SSPS dataset consists of application logs from software mimicking an embedded data collection device. The tested dataset contained 1,451,193 events broken up into 404 trace files. Our tool ran on this dataset in 0.88 seconds and used 14.1 MB of memory. The algorithm mined relations that established the sequential nature of the main application with no incorrect rules.

The SSPS system executes a periodic process with three phases: acquisition, processing, and communication. Data acquisition is accomplished using the *dd* utility to copy random data to a file. There are two phases of data processing. The first creates a checksum for the data using *cksum* and the second compresses the data using *bzip2*. The communication phase consists of using *scp* to transfer the compressed file to another host on the network. After the communication phase, the process is made pseudo-periodic by sleeping for a period.

The system logs an event between each phase, and between the two parts of the processing phase. This complicates log comprehension, as each event must serve as both the end of the previous phase and the beginning of the next. The trace also contains *anomalies*, in that the communication phase is sometimes absent. These issues are illustrated in Figure 2.15, which shows the phases of execution and their relationship to the events in the trace. In the figure, event ***D*** and *scp* interval are highlighted because they are absent from anomalous traces.

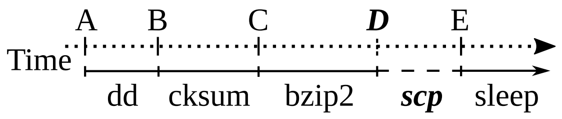


Figure 2.15: SSPS events and application phases

Another challenge of the SSPS dataset was that it was broken up into 404 separate traces that mostly contained events related to interrupt handling. Each trace contained, on average, only 8.6 events related to phases of the application. This meant that a single trace usually did not contain events from two full cycles of the main loop. These traces were non-contiguous, so events were missing between them and they could not simply be concatenated together and treated as one trace.

The intervals that were meant to be derived from the event sequence were the phases of execution of the main program and the pattern of an interrupt firing, its ISR starting, its ISR exiting, and the interrupt returning control to the operating system. Our algorithm mined relations that established the sequential nature of the main application with no incorrect rules. It also perfectly captured the relations pertaining to each interrupt occurring, being handled, and exiting.

The most interesting aspect of the mined relations from the SSPS dataset was that some of the rules defined the BJI of a phase rather than its duration. This highlighted a problem *with the dataset* rather than with our algorithm. We discovered that the system that generated the logs had an average CPU utilization of 84%. As this is above the theoretical safe limit for a real-time system of 69% (see Section 2.8.2) we should expect some tasks' BJIs to become shorter than their durations.

LANL Dataset

The System Call Logs with Natural Random Faults (LANL) dataset consists of application logs from a simulation of an automotive cruise-control application on a CPU under ionizing radiation bombardment [173]. The dataset contains faults from placing the SoC in the path of a high energy neutron beam at the LANSCE facility in New Mexico, USA. The dataset contained 100,000 events. Our tool ran on this dataset in 0.06 seconds and used 12.6 MB of memory.

The nfer specification for the LANL dataset defines request-response behavior for sensors, controllers, and actuators and then defines nominal and off-nominal relations between them. The cruise-control application that generated the dataset executes in a polling loop, but most events occur in response to external factors. The intervals that are meant to be derived from the event stream are the request-response patterns, where the response may be nominal or off-nominal.

The cruise-control application used for the LANL dataset consists of communicating components: a speed sensor, brake and accelerator actuators, and cruise controllers. The speed sensor is simulated and sends randomized speed readings to a configurable number of cruise controller functions which then command the brake or accelerator actuators to correct for any changes. Multiple cruise controller functions operate in parallel and their actuator commands are compared against one another as a form of error correction. The ionizing radiation present during system operation causes anomalies in the calculated actuator commands which result in changes to the trace.

The nfer specification for the LANL dataset defines periods of activity for each component and then defines the nominal and off-nominal behavior during those periods. During the period of activity for the speed sensor, the nominal behavior is for the speed value to be sent and no off-nominal behavior is defined. During the period of activity for the actuators, the nominal behavior is for a unanimous response to be received after a request to the controllers for commands is sent and the off-nominal behavior is for a non-unanimous or non-quorum response to be received. During the period of activity for the controllers, the nominal behavior is for the correct actuator to acknowledge receipt after a controller sends its command and the off-nominal behavior is for an incorrect actuator to acknowledge receipt of the command.

Our algorithm found every relation describing nominal behavior for the speed sensor and actuator components, but was unable to find off-nominal behavior relations. This is not surprising, as the faults that are part of the off-nominal behavior occur infrequently in the trace. As a result, the relations that include events from faults do not have enough support to be mined.

More surprisingly, many nominal behavior relations describing the controller functionality were missing from the mined rules. The only relation we mined from the controller behavior was from two events that appeared sequentially in the source code with only interprocess communication occurring between them. We discovered that the missing rules were explained by the parallel execution of multiple controllers. This design caused events to appear in non-deterministic order.

Figure 2.16 shows the problem of the parallel execution of multiple controllers in the LANL dataset. The events from every controller shared the same names, so it became impossible to find the correct relations without having a way to differentiate between them. We discovered that the controllers logged their process identifiers as data parameters in the trace, so we would be able to differentiate between the processes if we supported the notion that data parameters should be equal, as in the case of [191]. For now, we suggest that users choose unique event names per process instance. We plan to incorporate data parameters into our algorithm in future work.

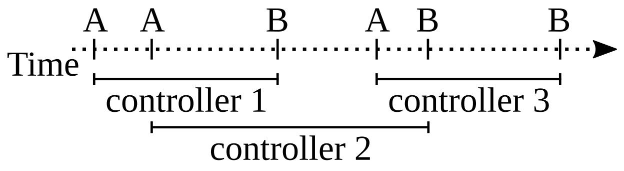


Figure 2.16: LANL parallel controller execution

MSL Dataset

The MSL dataset consists of telemetry in the form of EVRs received from the Curiosity rover on Mars. The events in the MSL logs are generated by the rover when commands are executed from daily activity plans uploaded by controllers on Earth. The logs cover a period of about 60 days and are filtered only to contain the events relevant to the available `nfer` specification, written for the case study in Section 2.6. The dataset contained 49,999 events. Our tool ran on this dataset in 0.64 seconds and used 45.7 MB of memory.

The `nfer` specification for the MSL dataset defines situations where errors that have been reported by the spacecraft can safely be ignored. We specifically analyzed one set of rules that defines a benign race condition where the routine servicing a radio reports missing telemetry due to thread starvation. The intervals that are meant to be derived take the form of a command being *dispatched* and later *completing*. The same pattern of *dispatch* **before** *complete* follows for 464 different types of commands in the trace, all intermingled together.

Our algorithm found 117 pairs of *dispatch* **before** *complete* relations for the same command. Encouragingly, no *complete* **before** *dispatch* rules were mined for the same command, meaning that our assumption that the duration should be shorter than the BJI held throughout.

To assist in assessing the relatively large number of rules produced from mining the MSL dataset, we developed a technique for grouping events based on the mined rules in which they appear. To group events, we compute a forest where each tree is made up of events that share rules. Algorithm 6 shows how to compute such a forest. In the algorithm, Groups is a map from event names in the trace alphabet $\mathcal{I}$ to group numbers. On Lines 5 through 7, each event name in the alphabet is iterated over and assigned a group if it does not already have one. Then, on Lines 8 through 12, all rules that refer to that event name are found, and both sides of the relation are assigned to the same group (one will be *name*). In this way, we organized the 937 event names from the MSL dataset into 90 groups with more than one event.

Algorithm 6 Event Name Grouping Algorithm

```
1:  procedure GroupEventsByRules($Specification$)
2:      $nextgroup \leftarrow 1$
3:      $Groups \leftarrow [\ ]$
4:      for $name \in \mathcal{I}$ do
5:          if Groups[$name$] is not set then
6:              Groups[$name$] $\leftarrow nextgroup$
7:              $nextgroup \leftarrow nextgroup + 1$
8:          for $rule \in Specification$ do
9:              if $rule$ refers to $name$ then
10:                 Groups[$rule.leftside$] $\leftarrow$ Groups[$name$]
11:                 Groups[$rule.rightside$] $\leftarrow$ Groups[$name$]
12:         end for
13:     end for
14:     return $Groups$
```

A scientist at JPL examined the mined rules from running our algorithm on the MSL dataset. He found that the rules captured some useful information about how the spacecraft behaves, and we were able to use his analysis to verify that a sequential task was correctly identified. For example, a set of rules successfully describes a periodic activity that loads a schedule table for the Rover Environmental Monitoring Station (REMS) instrument that monitors Mars' weather. The table tells the REMS instrument what data to collect and consists of a sequence of six commands. Our algorithm found relations between *dispatch* and *complete* events for each command, but it also found relations for the different commands in sequence.

2.9 Related Work

An earlier effort to develop a telemetry comprehension tool is described in [108], which provided a Scala DSL for writing a subset of the specifications shown in this article. That work was based on a still earlier effort using the rule-based system LogFire [107] for analyzing telemetry streams, as described in [109]. Although rule systems are strongly related to `nfer`, they are not suited for expressing minimality constraints for optimization purposes, as discussed previously. Other rule systems include Drools [75], Clips [52], and Jess [121].

Interval logics are common in the planning domain. Allen formalized his algebra [11], which has come to be known as ATL, for modeling time intervals. He argued that it was necessary to model relative timing with significant imprecision, and proposed his algebra's use in planning systems [12]. Many other planning languages have been proposed which rely on these same concepts, including PDDL [162] and ANMLite [42]. The concepts introduced and formalized by these interval logics are useful for modeling telemetry data, but the languages themselves have been principally designed for planning, not verification. Some efforts have been made to adapt them to that role, however. An effort is described in [205], where the suitability of the ANMLite system for verification was evaluated, with some positive results, but it was ultimately concluded that the solver techniques were not yet mature enough to be useful. A translation from LTL to PDDL is described in [10] as a means to leverage PDDL's solver for verification. Conversely, [196] defines a translation of a modified ATL to LTL for monitoring. It is concluded, however, that this approach is impractical since the generated monitoring automata become too large, even for small ATL formulas. Instead, they introduce a simple algorithm for that purpose using a state machine for each relationship. Other interval logics have been designed specifically for verification purposes, such as Interval Temporal Logic (ITL) [170], the Duration Calculus (DC) [105], and Graphical Interval Logic (GIL) [69].

Our work has strong similarities to data-flow (data streaming) languages. A recent example is QRE [14], which is based on regular expressions, and offers a solution for computing numeric results from traces. QRE allows the use of regular programming to break up the stream for modular processing, but is limited in that the resulting sub-streams may only be used for computing a single quantitative result, and only using a limited set of numeric operations, such as sum, difference, minimum, maximum, and average, to achieve linear time (in the length of the trace) performance. Our approach is based on Allen logic, and instead of a numeric result produces a set of named intervals, useful for visualization (and thereby systems comprehension). Furthermore, data arguments to intervals can be computed using arbitrary functions.

Runtime verification [149, 86] can generally be defined as the discipline of constructing monitors for analyzing systems executions. Assuming the type $\mathbb{E}$ of events, and some data domain D, a monitor can abstractly be considered as a function that takes a set of traces as an argument and returns a data value of type D. That is, M has the type $M : 2^{\mathbb{E}^*} \rightarrow D$. In most runtime verification systems a monitor processes a single trace. As such `nfer` can be seen as a runtime verification tool, processing a single trace and returning a set of intervals: $D = 2^{\mathbb{I}}$. Traditionally, however, runtime verification tools analyze traces in order to provide a Boolean verdict ($D = \mathbb{B}_2$), or some value in a simple extension of the Boolean domain [31], indicating whether a trace satisfies a specification or not (or the result can be unknown). Such systems include e.g. Eagle [21], MOP [163], Orchids [100], Ruler [23], TraceContract [22], LTL$_3$ [33], MarQ [194] (based on QEA - Quantified Event Automata [20]), LogFire [107], Larva [53], RiTHM [175], JUnitRV [63], MMT [64], MonPoly [26], and DejaVu [110]. Runtime verification systems have been developed which aggregate data as part of the verification [90, 25]. A system such as LOLA [59] computes general data streams from input data streams, and is in this sense more general than the Boolean verdict systems. Systems such as Ruler and LogFire produce sets of facts from traces, which is also more general than Boolean verdicts. Statistical model checking [146] is an approach collecting statistical information about the degree to which a specification is satisfied on multiple traces. In specification mining [83, 192] the user provides no specification. Instead it is learned from a collection of nominal executions.

The problem of mining patterns based on Allen's interval relations from a multivariate interval series has been thoroughly studied. Kam et al. looked for nested combinations of ATL patterns in an interval series [124]. Höppner used Allen's transitivity law in a sliding window to reduce the number of candidate patterns [115]. Likewise, De Amo et al. constrained candidate ATL patterns using regular expressions [62].

Our work in Section 2.8 is closely related to the field of specification mining using dynamic inference. Although we do not seek to mine general property specifications, we are interested in what Dwyer et al. called *response* and *precedence* patterns [78]. Yang et al. proposed solutions to the scaling problems of dynamic property inference in their Perracotta tool [235]. Ernst et al. introduced the popular Daikon tool that uses a library of patterns to efficiently check for program invariants [83]. Le Goues and Weimer found that they could reduce false positives in the mined properties by incorporating trustworthiness metrics [145]. Reger et al. introduced parametric specification mining using Quantified Event Automata (QEA) [191] and later proposed support for imperfect traces [193]. Lemieux et al. supported user-defined Linear Temporal Logic (LTL) patterns and imperfect traces in their Texada tool [147]. Cutulenco et al. efficiently mined TREs using a matrix of pattern automata [57].

A related and widely explored topic is *sequential pattern mining*, which seeks to find frequent subsequences of events in a database of sequences [160, 167]. Many sequential pattern mining works differ from ours because they seek to find arbitrary length patterns, because they expect the patterns to be frequent, and because the duration of the patterns is limited. Agraval and Srikant are often credited with introducing the idea, and proposed a set of Apriori-based algorithms for finding such consecutive subsequences [6]. Mannila et al. used sliding-window algorithms with pattern matching automata to mine frequent episodes, including serial episodes, from event traces [157]. Han et al. introduced a series of algorithms to reduce the search space of such algorithms by using tree-based data structures and by projecting subsequences into smaller databases [104, 177]. This work culminated in the well-known CloSpan algorithm by Yan et al. to find *closed* sequences, meaning sequences where no supersequence exists with the same support [234].

Some existing sequential pattern mining techniques have focused on performance. Zaki proposed Sequential PAttern Discovery using Equivalence classes (SPACE), which complexity improvements using combinatorial properties to decompose the problem using lattice search techniques [237]. Ayres et al. used a bitmap representation in their Sequential PAttern Mining (SPAM) tool to achieve improved performance over SPACE, but with a prohibitive increase in its memory requirements [16]. Wang and Han introduced their BI-Directional Extension (BIDE) tool to mine frequent closed sequences and achieved an order of magnitude improvement in speed over CloSpan [225]. They measured linear execution time and memory scalability in the size of the trace in their experiments. We are not, however, interested in mining frequent or closed sequences.

Ding et al. introduced a variation called *repetitive pattern mining* which more closely resembles our work [70]. Repetitive pattern mining considers infrequent patterns of arbitrary length that may repeat in a trace. Our work achieves improved time and space complexity over theirs but our application is more specialized to finding patterns of length two in real-time system traces.

2.10 Conclusion

We have introduced the `nfer` rule-based formalism and system for inferring event stream abstractions. The problem has been inspired by actual planetary space mission operations, specifically the Mars Curiosity rover. The result of applying an `nfer` specification to an event stream is a set of intervals: named sections of the event stream, each including a start time, an end time, and a map holding data. Intervals are formed from events and other intervals, forming a hierarchy of abstractions. The result may be visualized or queried, and can generally help engineers to better comprehend the contents of an event stream. Rules may be mined from historical data to assist users or to validate specifications. The `nfer` system is implemented in both Scala and C, with an external DSL for expressing rules, in addition to internal Scala and C DSLs (APIs). APIs are also available in the popular programming languages R and Python. The system has been shown to scale well, aided by simple algorithmic enhancements without much loss of precision.

Chapter 3

Online Distributed Anomaly Detection

3.1 Introduction

Anomaly detection is a form of Runtime Verification (RV) where nominal behavior is typically learned from the statistical analysis of historical data. In many cases, anomaly detection concerns continuous signals rather than the more typical event series in other forms of RV. However, both terms may be applied to the detection of unexpected behavior in either continuous or discrete data that may be a symptom of a failure.

Failures may be caused by logical faults, which can be avoided with good design principles, or execution faults, which can be difficult or impossible to detect at design time [153]. Execution faults may be caused by software errors such as memory leakage or deadlocks, or hardware errors like degraded sensors. Malicious attacks may also cause execution faults, exploiting weaknesses in hardware and software to cause unpredictable behaviors. As safety-critical embedded systems become more ubiquitous and connected, such attacks become more scalable and realistic [219]. Examples of attacks on embedded systems include disabling a car's braking system [161] and injecting a fatal dose of insulin in an insulin pump [158].

Detection systems must be able to identify faults and attacks quickly and accurately for their results to be useful. If an anomaly is missed it cannot be corrected, and if an anomaly is detected too late, the correction may be unable to prevent a failure. To use an analogy, anomalies such as faults and attacks can be thought of as diseases in a system. To prevent a disease from causing a system failure, it is necessary to understand the disease's symptoms, carefully watch for them, and act quickly if any are detected.

In this chapter, we introduce a taxonomy of anomaly symptoms and a framework for low-latency online detection of these anomaly symptoms called Palisade. Our system is designed to monitor remote, real-time embedded systems, and to provide a unified mechanism for responding quickly to faults and attacks. Palisade is also designed to be extensible to facilitate the incorporation of new anomaly detection algorithms to detect the symptoms of unforeseen anomalies. Among other detectors, Palisade includes an integration with `nfer`.

The main contributions of this chapter are:

- We describe a comprehensive taxonomy of symptoms of anomalies that may occur in embedded systems and give examples of instances in the literature where they occur. This taxonomy is a valuable resource for anomaly detection work as it supplies a shared language with which researchers and practitioners can discuss their capabilities and requirements.
- We propose Palisade, a data streaming framework that supports online anomaly detection for embedded systems. We present its software architecture, design choices, included anomaly detectors, and two evaluations of its detection latency and extensibility.
- We evaluate the applicability of Palisade through two case studies: one using real data from an autonomous car and a second using data generated from an autonomous driving development platform. We show that, by integrating different anomaly detectors, Palisade is able to detect more anomalies than a stand-alone detector.

Palisade deviates from conventional anomaly detection frameworks in that it is designed to monitor remote, embedded systems and to provide online verdicts with low latency. Many anomaly detection algorithms have been proposed in recent years but the vast majority are designed to operate offline, by analyzing recorded traces [8, 44, 176, 19, 9, 184]. Offline anomaly detection is useful for post mortem analysis of problems, but not to prevent failures at runtime. To monitor remote systems, events and readings must be transmitted to detection algorithms using a data streaming architecture. Palisade uses the publish-subscribe interface from the Redis in-memory database [190].

Our contributions improve on the state-of-the-art of both taxonomies of anomalous behavior and anomaly detection frameworks. We propose a finer grained and more extensive taxonomy of anomalies from prior works, including symptoms of anomalies in an event series. Ours is also the first framework that unifies many algorithms into an online anomaly detection system that can be applied to remote, embedded systems.

The rest of this chapter is organized as follows. Section 3.2 describes anomaly symptoms in both continuous signals and event series. Section 3.3 introduces and discusses the Palisade architecture. Sections 3.4 and 3.5 present the two case studies and performance results. Section 3.6 discusses a number of ways to evaluate the framework. Section 3.7 summarizes prior work related to this chapter. Section 3.8 concludes the chapter.

3.2 Anomaly Symptoms

This section logically groups the anomaly symptoms that are present in the observable outputs of a system. These symptoms represent the realization of a perturbation in an internal, unobserved state machine. These symptoms do not prove an anomaly by their mere presence, but an anomaly may cause one or more symptoms, hence the disease-symptom analogy. The list in this section is not exhaustive, but categorizes common anomaly symptoms.

This taxonomy relates to Mitre's Common Attack Pattern Enumeration and Classification (CAPEC) [165], in that both CAPEC and this taxonomy can be used to classify capabilities and behaviors. They differ substantially in that CAPEC describes possible attacks, while this section simply describes *symptoms* of attacks or other, non-malicious anomalies.

Many of the symptoms defined in this section are expressed in relation to a set of parameters. Examples include the constant factor c expressing the spike height for spike symptoms, or the difference ℓ expressing the difference required to define a level change symptom. For a given system, a user can determine these constants using a system simulation based on prior knowledge, or traditional parameter-finding approaches such as grid search [36, 89].

3.2.1 Notation Used in this Section

The arithmetic mean, sometimes just called the mean, of n values $\sigma = \langle x_1, \cdots, x_n \rangle$ is $\bar{\sigma} = \frac{1}{n}\Sigma_{i=0}^{n} x_i$. The mean of a sequence σ is denoted $\bar{\sigma}$. The variance of n values $\sigma = \langle x_1, \cdots, x_n \rangle$ is $var(\sigma) = \frac{1}{n}\Sigma_{i=0}^{n}(x_i - \bar{\sigma})^2$ and the standard deviation (stdev) is the square root of the variance $stdev(\sigma) = \sqrt{var(\sigma)}$. The absolute value of a number n, denoted $|n|$ is n if $n \geq 0$ or $-n$ if $n < 0$. We use Iverson Bracket notation $[Q]$ for some Boolean condition Q to denote $[Q] = \begin{cases} 1 & \text{if } Q \text{ is true} \\ 0 & \text{otherwise} \end{cases}$. The standard normal distribution is given as the probability density function $N(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$. The general normal distribution is the standard normal distribution with the domain stretched to a specified stdev and translated to a specified mean. We denote the general normal distribution as $\mathcal{N}(x|m, s) = \frac{1}{s} N(\frac{x-m}{s})$. The *Discrete Fourier Transform (DFT)* of a time series is a sequence of the same length of the frequency components of the time series. Given a time series σ with length N, its DFT X is defined by $X_k = \Sigma_{n=1}^{N} \sigma_n e^{-\frac{2\pi i}{N} k(n-1)}$. We write $\mathcal{F}(\sigma)$ to denote the DFT of σ.

3.2.2 Continuous-Signal Anomaly Symptoms

For the purposes of anomaly symptoms, we define a continuous signal as a digitally sampled signal with a constant sample rate, represented here as a time series of type $\mathbb{R}^*$. This signal is expected to be the result of readings from a single sensor, not an amalgamation of many sources. The constant sample rate means that the sample time of each value is known from its index in the time series.

Spikes and S-waves

We define a Spike (Figure 3.1a) as a subsequence of contiguous samples that lie farther than a given number of standard deviations from the current mean of the signal. To account for signals with means that change over time, we consider the distance to the mean of a *window* of samples prior to the subsequence.

Definition 1 (Spike)**.** *Given a time series $y \in \mathbb{R}^*$, indices p and q, a window size $n \in \mathbb{N}$, and a constant factor $c \in \mathbb{R}$, the subsequence $y_{[p+1,q]}$ exhibits a Spike if for all y_t where $p < t \leq q$ it holds that $|y_t - \bar{y}_{[p-n,p]}| > c\ (stdev(y_{[p-n,p]}))$.*

We define S-waves (Figure 3.1b) as spikes with an additional deviation in the opposite direction immediately following the spike. S-waves can mimic spikes if the counter-spike is sufficiently dampened.

Example: A flooding attack over a vehicle Controller Area Network (CAN) may falsely indicate that the collision prevention system issued a command to engage the brakes [164]. Such an attack falls under the category of ⟨CAPEC-125: Flooding⟩ [165] and could be detected by monitoring for Spike anomalies in the volume of CAN packets.

Drifting

A Drift (Figure 3.1c) is a slow movement of the signal mean over a period of time. We consider only linear drift here; logarithmic and sub-linear drifts are rare, and higher order drifting encroaches on the definition of level changes or spikes. Mathematically, a continuous signal y is offset by tc, where t is the time index and c is a constant representing the slope of the drift.

Definition 2 (Drift)**.** *Given a time series $y \in \mathbb{R}^*$, indices p and q, a nominal version of the time series $\hat{y} \in \mathbb{R}^*$, and a slope $c \in \mathbb{R}$, a subsequence $y_{[p,q]}$ exhibits linear Drift if for all y_t where $p \leq t \leq q$ it holds that $y_t = \hat{y}_t + tc$.*

Example: An infrared combustible sensor, when functioning over the operational temperature limit, may drift or fail [195]. Such a failure could be detected by monitoring temperature readings for Drift anomalies.

Noise

Noise (Figure 3.1d), an expected component of any signal, is considered a symptom of an anomaly only when it is more pronounced than is typical. We define noise as a normally distributed offset around the true value of the signal.

Definition 3 (Noise)**.** *Given a time series $y \in \mathbb{R}^*$, indices p and q, some noisiness coefficient $n \in \mathbb{R}$, and a nominal version of the time series $\hat{y} \in \mathbb{R}^*$, a subsequence $y_{[p,q]}$ exhibits Noise if for all y_t where $p \leq t \leq q$ it holds that $y_t = \hat{y}_t + \mathcal{N}(0, n)$ where $\mathcal{N}(0, n)$ is a standard normal distribution centered at zero with standard deviation n.*

Example: Compressed air in truck brakes may generate acoustical interference and cause metallic friction noise from track vehicles in ultrasonic sensors [201]. Brake failure could be detected by correlating Noise anomalies in ultrasonic sensors with air brake usage.

Clipping

We define Clipping as a loss of data at the extrema of a signal range (Figure 3.1e), where a signal is of a higher amplitude than is supported by the sensor or transmission medium. Thus a clipped signal is indicated by a large proportion of samples at the maximum or minimum extent of the sample medium.

Definition 4 (Clipping)**.** *Given a time series $y \in \mathbb{R}^*$, indices p and q, maximum and minimum possible sample values $y_{max}, y_{min} \in \mathbb{R}$, and a threshold $\epsilon \in \mathbb{R}$, a subsequence $y_{[p+1,q]}$ exhibits Clipping if $(\Sigma_{p<t\leq q}([y_t = y_{max}] + [y_t = y_{min}])/(q-p)) > \epsilon$.*

Example: A partially blinding attack on a camera of a vehicle by emitting light can hide objects [181]. This light can exceed the input range of the camera and would appear as clipped. This attack is an example of ⟨CAPEC-607: Obstruction⟩ [165]. Such blinding light attacks could be detected by monitoring for Clipping anomalies.

Loss

While Loss (Figure 3.1f) may more typically refer to high noise levels making it difficult to decode a signal, here we use loss to indicate a complete loss of a signal. Although trivially an anomaly, a total loss of signal may be a symptom of temporary network disruption without any more dangerous cause. We represent a total loss of signal as a sudden transition to a fixed sample value. This can be observed as a special case of Clipping, where the extrema of the signal are identical for a short time.

Definition 5 (Loss)**.** *Given a time series* $y \in \mathbb{R}^*$*, indices* p *and* q*, and a threshold* $\epsilon \in \mathbb{R}$*, a subsequence* $y_{[p+1,q]}$ *exhibits Loss if* $(\Sigma_{p<t\leq q}([y_t = y_{t+1}])/(q-p)) > \epsilon$*.*

Example: An attack sending a large volume of request messages over the J1939 protocol increases the computational load of the recipient ECU until it is not able to perform regular activities like transmitting periodic messages [171]. Such an attack is an example of ⟨CAPEC-125: Flooding⟩ [165] and could be detected by monitoring CAN traffic for Loss anomalies.

Smoothing

We define Smoothing to be a reduction in the short term variance of a signal compared to recent history. Smoothing (Figure 3.1g) is the rarest of the symptoms presented here, with few natural causes.

Definition 6 (Smoothing)**.** *Given a time series* $y \in \mathbb{R}^*$*, an index* t*, a constant* $k \in \mathbb{N}$ *representing how far back the recent historical signal variance should be considered, and the factor threshold* $\tau \in \mathbb{R}$ *at which the signal is considered smoothed, a subsequence of* n *samples* $y_{[t,t+n]}$ *exhibits Smoothing if* $var(y_{[t,t+n]}) < var(y_{[t-(nk)-1,t-1]})\tau$*.*

Example: In an attack of a control system, the attacker may observe and record sensor readings and then continuously repeat the recorded values during the attack [166]. This is an example where the sensor values are smoothed. Such an attack falls under the category of ⟨CAPEC-148: Content Spoofing⟩ [165]. Such spoofing attacks could be detected by monitoring sensor readings for Smoothing anomalies.

Amplification

Amplification (Figure 3.1h) is a simple gain on the target signal. For amplification of an original signal we multiply every sample by some factor.

Definition 7 (Amplification)**.** *Given a time series* $y \in \mathbb{R}^*$*, an index* t*, the magnitude threshold of amplification* $\alpha \in \mathbb{R}$*, and an unamplified time series* $\hat{y} \in \mathbb{R}^*$*, a sample* y_t *exhibits Amplification if* $y_t > \alpha \hat{y}_t$*.*

Example: Analog to Digital Converters (ADCs) can be attacked by amplifying analog signals past the dynamic range of the device. These attacks can obscure other malicious behavior and damage hardware [39]. This type of attack is an example of ⟨CAPEC-153: Input Data Manipulation⟩ [165]. It could be detected by monitoring the analog signal for Amplification anomalies.

Level Change

A Level Change (Figure 3.1i) symptom is observed when the mean of a signal changes in a short period and then remains consistent at the new level. Slower changes may fall under drifting.

Definition 8 (Level Change)**.** *Given a time series* $y \in \mathbb{R}^*$*, an index* t*, a window size* $w \in \mathbb{N}$*, an acceptable minimum level change threshold* $\ell \in \mathbb{R}$*, and a minimum number of samples the mean change must persist* $n \in \mathbb{N}$*, a subsequence* $y_{[t,t+w-1]}$ *exhibits a Level Change if* $|\bar{y}_{[t+w,t+w+n]} - \bar{y}_{[t-n-1,t-1]}| > \ell$*.*

Example: An attack that increases the amount of code execution will increase the power consumption of the Central Processing Unit (CPU), which can be observed as a Level Change [168, 169, 132]. Such an attack could be an example of ⟨CAPEC-175: Code Inclusion⟩, or ⟨CAPEC-242: Code Injection⟩ [165]. Many attacks with this profile can be detected by monitoring the power consumption of the CPU for Level Change anomalies.

Frequency Change

A Frequency Change (Figure 3.1j) occurs when the primary frequency of a signal changes over a short period. We say a Frequency Change occurs if the primary frequency in a sliding window moves more than some threshold over some time window.

Definition 9 (Frequency Change). *Given a time series $y \in \mathbb{R}^*$, an index t, a window size $w \in \mathbb{N}$, a function $P : \mathbb{R}^* \to \mathbb{R}$ which extracts the frequency of the highest peak from a DFT (denoted $\mathcal{F}$), a threshold $\epsilon \in \mathbb{R}$, and a minimum number of samples the frequency change must persist $n \in \mathbb{N}$, a subsequence of w samples $y_{[t,t+w-1]}$ exhibits Frequency Change if $|P(\mathcal{F}(y_{[t+w,t+w+n]})) - P(\mathcal{F}(y_{[t-n-1,t-1]}))| > \epsilon$*

It may be useful to consider more frequencies, but we restrict our definition to only consider the primary frequency for simplicity.

Example: An attack inserting a burst of light into a vehicle camera may change the tonal distribution (light frequency) of the captured images, resulting in misclassification [181]. This attack is an example of ⟨CAPEC-607: Obstruction⟩ [165] and it could be detected by monitoring captured images for Frequency Change anomalies.

Echo/Reflection

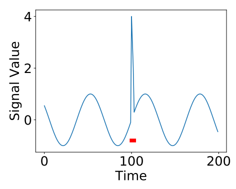
We consider an Echo (Figure 3.1k) to be a duplication of a previous series of samples on top of the underlying signal at a later position. A Reflection is identical to an Echo, except that the repeated signal is inverted.

Definition 10 (Echo). *Given a time series $y \in \mathbb{R}^*$, indices t and t', an echo length $e \in \mathbb{N}$, an echo coefficient (the factor at which the echo is played back) $q \in \mathbb{R}$, and the nominal form of the time series $\hat{y} \in \mathbb{R}^*$, if we consider the subsequence $y_{[t,t+e]}$ as the origin of the echo, the subsequence $y_{[t',t'+e]}$ exhibits Echo if $y_{[t',t'+e]} = \hat{y}_{[t',t'+e]} + y_{[t,t+e]} \times q$.*

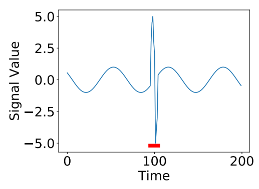
Example: A relay attack on the original signal sent from the vehicle LiDAR creates fake echoes and can make real objects appear closer or further than their actual location, thus affecting the mission planning [181]. This attack is an example of ⟨CAPEC-586: Object Injection⟩ [165]. Such an relay attack could be detected by monitoring the LiDAR signal for Echo and Reflection anomalies.

3.2.3 Event-Series Anomaly Symptoms

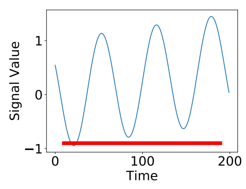
Symptoms of anomalies also appear in event series, which are defined as a sequence of discrete events rather than a continuous signal. An event series represents a trace of the execution of an automaton where each event describes a state transition. Event series differ from discretized continuous signals in that their events are not required to occur at regular time intervals and they may carry more complex data than only a single real value.



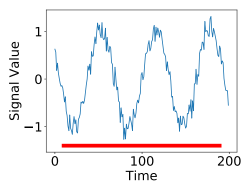
(a) Spike



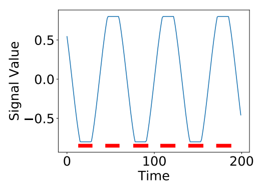
(b) S-Wave



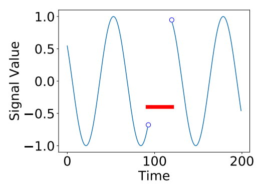
(c) Drifting



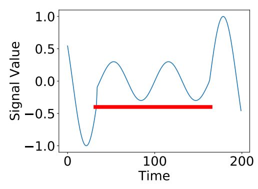
(d) Noise



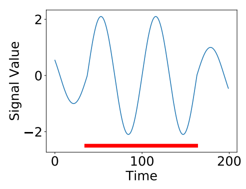
(e) Clipping



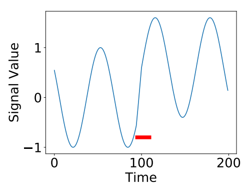
(f) Loss



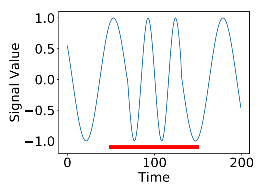
(g) Smoothing



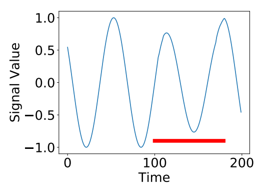
(h) Amplification



(i) Level change



(j) Frequency change



(k) Echo/Reflection

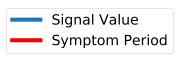


Figure 3.1: Time Series Anomaly Symptoms. The red line indicates the period of the described anomaly

Event Frequency Change

When events of the same name are periodic or semi-periodic, they have fairly consistent inter-arrival times and, by extension, frequency. When that frequency changes suddenly, it can be a symptom of a system anomaly. Similarly, when the frequency of all events in a trace change suddenly, it may be due to an anomaly.

The inter-arrival time of an event is the difference between the clock times of successive events of the same name. It can be thought of as the *period* of the event.

Definition 11 (Inter-arrival Time)**.** *Given a trace $T \in \mathbb{T}$, an event name $\mu \in \mathcal{I}$, and a non-empty time interval defined by the end points $t_1, t_2 \in \mathcal{C}lock$ where $t_1 < t_2$, the inter-arrival time is given by the curried function interArrival* $: \mathbb{T} \to \mathcal{I} \times (\mathcal{C}lock \times \mathcal{C}lock) \to \mathcal{C}lock$*. Define a set S containing all the events between t_1 and t_2, $S = \{(\mu, t, \cdot) \in T : t_1 \leq t \leq t_2\}$.*

For all $(\mu, t, \cdot) \in S$, *interArrival*$(T)(\mu, (t_1, t_2)) = (\max t - \min t)/(|S| - 1)$.

Event frequency measures how often events occur in a given time span. It is the inverse of inter-arrival time.

Definition 12 (Event Frequency)**.** *given a trace* $T \in \mathbb{T}$*, an event name* $\mu \in \mathcal{I}$*, and a non-empty time interval defined by the end points* $t_1, t_2 \in \mathcal{C}lock$ *where* $t_1 < t_2$*, the event frequency of* μ *is given by the curried function eventFreq* : $\mathbb{T} \to \mathcal{I} \times (\mathcal{C}lock \times \mathcal{C}lock) \to \mathcal{C}lock$*. It is defined as eventFreq*$(T)(\mu, (t_1, t_2)) = 1/$*interArrival*$(T)(\mu, (t_1, t_2))$*.*

A sudden change in event frequency, then, is when the first derivative of event frequency is high. A rapid change in event frequency can be found by taking the difference between successive time intervals (or *windows*) in the trace. If the difference exceeds some threshold, then the change in event frequency may indicate an anomaly.

Definition 13 (Frequency Change)**.** *Given a trace* $T \in \mathbb{T}$*, an event name* $\mu \in \mathcal{I}$*, a window size* $w \in \mathcal{C}lock$*, and a threshold* $\epsilon \in \mathbb{R}$*,* T *contains an Event Frequency Change if* $\exists t_1, t_2, t_3 \in \mathcal{C}lock$ *such that eventFreq*$(T)(\mu, (t_1, t_2)) -$ *eventFreq*$(T)(\mu, (t_2, t_3)) > \epsilon$*.*

Example: Lin and Siewiorek introduced their Dispersion Frame Technique (DFT) to predict hardware failures [152]. From analyzing the logs of file servers, they observed that there exists a period of an increasing rate of intermittent errors before most hardware failures. Many such failures could be detected by monitoring error reports for Event Frequency Change anomalies.

Unexpected Event

Most traces only contain events with a limited vocabulary of event names. While events themselves are unique, due to their varying clock times, the event names are repeated many times. When an event occurs in a trace with a name that has not come earlier in the trace, it may be a symptom of an anomaly.

Definition 14 (Unexpected Event)**.** *Given a trace* $T \in \mathbb{T}$*, an Unexpected Event is an event* $(\mu, t_1, \cdot) \in T$ *such that* $\nexists(\mu, t_2, \cdot) \in T$ *where* $t_2 < t_1$*.*

By this definition, however, most events at the beginning of a trace will be considered unexpected. To solve this problem, we can restate the definition in terms of the probability that an event occurs.

Definition 15 (Improbable Event)**.** *Given a trace $T \in \mathbb{T}$ and a threshold $\epsilon \in \mathbb{R}$, an Improbable Event is an event $e \in T$ such that $\mathcal{P}(e \in T) < \epsilon$.*

Example: Bellovin reported receiving broadcast packets meant for local networks, requests to unused ports, and requests to unoccupied addresses over the public Internet at AT&T in his classic whitepaper [35]. These types of requests are examples of ⟨CAPEC-169: Footprinting⟩ [165] and they could be detected by monitoring network traffic for Unexpected Event anomalies.

Periods of Silence

A period of silence in a trace is a segment of time where no, or few, events occur. Events may occur more-or-less frequently during the operation of a system as different behaviors result in different patterns in the trace. However, nominal system behavior usually results in *some* events appearing. When a period occurs where no events appear in the trace, it may be a symptom of an anomaly.

Definition 16 (Period of Silence)**.** *Given a trace $T \in \mathbb{T}$ and a minimum number of events $\nu \in \mathbb{N}$, a Period of Silence is a non-empty time interval defined by the end points $t_1, t_2 \in \mathcal{C}lock$ where $t_1 < t_2$ such that $|\{(\cdot, t, \cdot) \in T : t_1 \leq t \leq t_2\}| < \nu$.*

The threshold for when a time interval is considered a period of silence varies from system to system. Some high priority tasks may monopolize system resources while not emitting any events. To solve this problem, we can restate the definition to specify a minimum length of the interval.

Definition 17 (Minimum Period of Silence)**.** *Given a trace $T \in \mathbb{T}$, a minimum number of events $\nu \in \mathbb{N}$, and a minimum length $\ell \in \mathbb{R}$ where $\ell > 0$, a Minimum Period of Silence is time interval defined by the end points $t_1, t_2 \in \mathcal{C}lock$ where $t_2 - t_1 \geq \ell$ such that the interval is a Period of Silence by Definition 16.*

Example: Missing log messages can indicate problems and failures in High Performance Computing (HPC) logs that are too large for humans to manually analyze [106]. These types of failures can be detected by monitoring logs for Periods of Silence anomalies.

Sampled Value Anomaly Symptom

When an event trace includes sampled values from a continuous signal, those sampled values may include the same trace anomalies defined in Section 3.2.2. An event trace is not sampled at a fixed rate like a time series, however. To test for sampled value anomaly symptoms, it may be necessary to extrapolate the values between samples to approximate a continuous signal.

There are several popular methods for recovering a continuous signal from irregular samples. These algorithms include, for example, the projections onto convex sets (POCS) method [212], the Wiley/Marvasti method [159], the Sauer/Allebach Algorithm (also called the Voronoi method) [200], and the Adaptive Weights Method [88]. These methods generally construct an auxiliary signal from some sample values and then obtain an initial approximation by applying a low-pass filter. The error between this approximation and the samples is then fed into an iterative algorithm which can recover the signal if the sampling density is high enough.

Example: Changes in byte frequency patterns in network payloads to the same host and port can be accurate predictors of network intrusions [226]. Such attacks can be detected by monitoring for Sampled Value Frequency Change anomalies.

3.3 Palisade Architecture

Palisade is motivated by the need to remotely detect a dynamic range of anomaly symptoms in an embedded system at the time they occur. We are further motivated by the desire to combine multiple anomaly detectors to leverage their different performance characteristics into a single, more reliable, detector. These motivations lead to the following requirements:

1. the anomaly detection must have low latency,
2. it must be easy to implement and maintain detectors,
3. the detectors must be able to be run on separate machines,
4. multiple detectors must be able to run in parallel on the same data, and
5. deployment of the system must be simple.

Requirement 1 is due to the need to respond to anomalies with enough time to mitigate their effects. Requirement 2 is important to any serious software framework, since it should always be a goal to reduce engineering costs. Requirement 3 allows Palisade to run on separate machines from the monitored system and to support anomaly detection appliances to plug into existing networks. It also facilitates horizontal scaling. Requirement 4 is related to Requirements 1 and 3, since operating multiple detectors in series over multiple machines would delay detection and require complex sequencing. Requirement 5 is to reduce the barriers to adoption of the system: if it is hard to install, no one will use it.

Palisade is designed as a set of distributed micro-services built around a data streaming architecture. These micro-services are implemented as three types of nodes: sources, processors, and sinks. Nodes are typically written in Python as Palisade provides a Python API to simplify their creation and maintenance. However, it is also possible to integrate existing tools written in other languages with Palisade. Figure 3.2 presents an overview of the Palisade architecture through a Unified Modeling Language (UML) information flow diagram [198]. Each node uses the publish-subscribe interface of the Redis data streaming infrastructure to receive and send data. We do not discuss the GUI in this chapter, as it is out-of-scope.

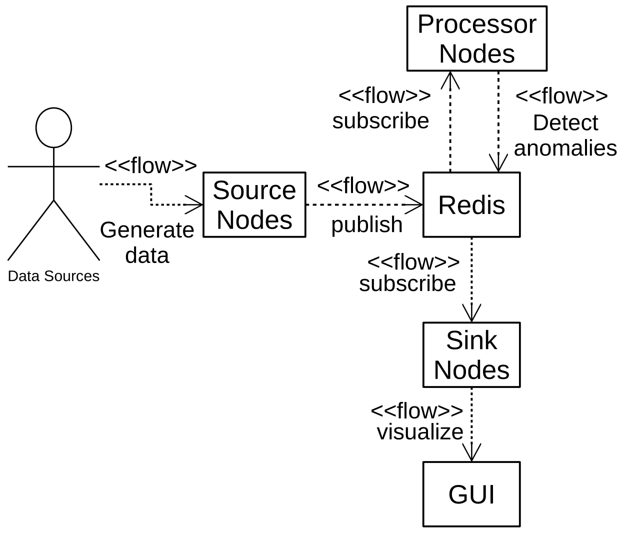


Figure 3.2: UML information flow of the Palisade architecture

3.3.1 Data Streaming

To support Requirements 1, 3, and 4, we needed to find a distributed streaming architecture to transport information between different nodes. We evaluated the latency of four streaming frameworks while considering their inclusion in the Palisade architecture. Latency is defined as the time difference between the instant data is generated by a source and the instant it is received by a processor or a sink [77]. As a result of this experiment we chose Redis as the data streaming architecture for Palisade.

We designed a set of experiments to measure the latency of four data streaming frameworks: Redis [190], RabbitMQ [189], Kafka [138], and NATS [174]. We used two first generation Raspberry Pis (single 700 MHz ARM6 core, 128 MB system RAM) for the subscriber and the publisher and a Raspberry Pi 2 (quad-core ARM Cortex-A7, 1G RAM) for the server. The clients and the server were synchronized using Precision Time Protocol (PTP), a network level time synchronization protocol capable of microsecond accuracy. We varied the transmitted message sizes (256 bytes, 1 KB, 100 KB, and 1 MB) and the publishing frequency (30 Hz, 60 Hz, and 100 Hz). Latency was measured by including the timestamp at which a message was sent within the message itself. The subscriber then noted the timestamp at which it received the message and subtracted the sending timestamp to find the latency. We ran each configuration of the experiment five times and extracted the average and standard deviation of the latency.

Table 3.1 shows the results for Redis latency. We note that mean latency increases as the packet size increases, as does the standard deviation. The throughput also increases when both frequency and packet size increase, reaching 6 MB/s at 1 MB and 100 Hz. Redis presents the lowest latency of the four data streaming frameworks. Refer to [77] for a complete comparison.

Table 3.1: Redis latency results in seconds

Freq. (Hz)/Packet size	256 B	1 KB	100 KB	1 MB
30	0.0125 ± 0.005	0.184 ± 0.005	0.0731 ± 0.035	0.315 ± 0.3
60	0.0218 ± 0.005	0.0213 ± 0.005	0.0461 ± 0.035	0.337 ± 0.3
100	0.0425 ± 0.005	0.0245 ± 0.005	0.0946 ± 0.035	0.365 ± 0.3

REmote DIctionary Server (Redis) is an open-source, in-memory, key-value database that provides a publish-subscribe interface. Redis clients publish data to channels using the REdis Serialization Protocol (RESP), and subscribers receive data in the same order it was published. Redis also supports integration with on-disk databases. Moreover, Redis has low memory consumption; in a 64-bit system, 1 million keys (hash values), representing an object with five fields, use around 160 MB of memory [190]. Redis provides a master-slave replication mechanism in which slave server instances are exact copies of master servers.

To reduce the network round trip latency, Redis implements *pipelining*, making it possible to send multiple commands to the server without waiting for individual replies [190]. These replies are instead batched together into a single response.

3.3.2 Data Format

Formatting the data transmitted between Palisade nodes requires choices to be made between the relative importance of the requirements listed in Section 3.3. If Requirement 2 (implementation) and Requirement 5 (deployment) are more important, then a textual format is preferable due to its increased interoperability and human readability. If Requirement 1 (latency) is the most important one, then a simple, offset-based binary format is better because of its lower bandwidth requirements and parsing costs. Palisade handles this conflict by supporting both JavaScript Object Notation (JSON) and a custom binary format for data transmission between nodes.

JSON is a standardized data format [79] with support in most programming languages [123], which makes it easy to consume and produce messages compatible with Palisade from different tools. JSON is a textual format, meaning the data is represented as sequences of characters and, as such, must undergo some processing to convert to-and-from internal program state. While this does increase the processing requirements of JSON formatted data, it has the benefit of being human readable which substantially eases debugging. Formatting with JSON also means that new parameters may be added without breaking backward compatibility since a parser will ignore JSON object keys that aren't expected.

In contrast, an offset-based binary format needs to be strictly specified and any changes require updates to producers and consumers alike. Such a binary format has the advantage of transmitting less data and does not require parsing since its fields may be found by their memory offsets (like a C struct). Some binary formats, like Google's Protocol Buffers, are more complex and allow some modification without updating all uses. However these features always incur additional costs that must be considered. For example, it is only possible to extend a Protocol Buffer message if field numbers were previously allocated for that purpose [71].

To support both sampled continuous value signals and discrete event series data, Palisade includes JSON formats for both time series and event series content. Palisade only supports a binary format for time series data. The time series binary format also enables integration with high throughput data sources such as a logic analyzer. Palisade does not currently support more complex binary formats that attempt trade-offs between extensibility and latency as there are no current use cases for such a format.

Figure 3.3 shows different JSON formats for event and time-series data. Time-series data (Figure 3.3a) contains the initial `timestamp` for the received message as well as a `frequency`, which is used to calculate the timestamp for each datum (by using the frequency and the timestamp of the first data). For the event-based format (Figure 3.3b), `timestamp` represents the instant in which the event is produced and the `data` field can be composed of arbitrary subfields (also in JSON format).

Figure 3.3c represents the binary format for time series. There are three fields in little-endian format. The first field is the millisecond component of the message time, followed by the two magic bytes set to zero, followed by the seconds portion of time, and the data in 16-bit signed two's complement integer format. In Figure 3.3c, there are 100 samples, which makes the data field have 200 bytes (100 integers, 2 bytes each).

```
{
  "timestamp": 12345.5678,
  "data": [0.3, 0.1, −0.5],
  "frequency": 100000
}
```

(a) Time-series JSON format.

```
{
  "timestamp": 12345.5678,
  "data": {
    "somefield": 13.3,
    "someotherval": "test"
  }
}
```

(b) Event-series JSON format.

```
0x5D03      // 861
       milliseconds
0x0000      // magic bytes
0x00100000 // 256 seconds
<data> // 200 bytes = 100
       16−bit signed ints
```

(c) Time-series binary format.

Figure 3.3: Examples of data formats used in Palisade

3.3.3 Source Nodes

Source nodes are responsible for streaming data into Redis. These nodes select and transmit data from a database, file, or an embedded system. Examples of source nodes are one that reads data from a relational database instance, a node that reads a Comma Separated Value (CSV) file, and a User Datagram Protocol (UDP) sniffer that reads packets from the network. Data might include system log entries, aggregate network states, or commands from an autonomous driving stack. Usually, a source node should read data from some data source, change it to JSON or binary Palisade formats, and publish it to a Redis channel.

3.3.4 Processor Nodes

Processors in Palisade are responsible for detecting anomalies and forwarding the results to sink nodes (see Section 3.3.5). Each processor implements a different anomaly detection algorithm.

All processors are sub-classes of the abstract *Processor* class. A processor object is instantiated by the *ProcessorLauncher* class, which associates the processor with an instance of the *DataSource* class, which supplies data from either Redis or an offline file source for unit testing purposes.

Individual processors must inherit from the *Processor* class and implement an interface used by the *ProcessorLauncher* to pass data from the *DataSource*. The relevant methods required are *configure* (which collected metadata) and *test_ on_ data* (which is invoked when new data is ready from the *DataSource*). Other optional methods that may be implemented by processors are: *begin* (called on startup), *end* (called on shutdown), *load_ model* (called if a model is specified in *configure*) and *train* (used to build models where applicable).

All non-source nodes subscribe to a channel, named `command`. This is motivated by Requirements 3 and 5, that deployment must be distributed and simple. Without this channel, each Palisade node would need to be individually controlled from a local interface. The `command` channel supports the control commands for nodes (such as restart and info (a status command)) the are used for general system maintenance.

Once a detector finds an anomaly, it publishes the timestamp at which the anomaly occurred, a note about its cause, and the source channel of the anomaly to a specific Redis channel. Figure 3.4 shows an example of the JSON data format for anomalies in Palisade. The `timestamp` field contains the time at which the anomaly occurred. The `anomaly` field is a measure of the confidence ($c \in \mathbb{R} : 0 < c \leq 1$) of the detector that an anomaly has occurred, where 1 represents 100% confidence. The `note` field is a textual description of the anomaly, and `channel` contains the Redis channel in which the anomaly happened.

```
{
    "timestamp": 12345.5678,
    "anomaly": 1,
    "note": "what happened",
    "channel": "input channel name
        "
}
```

Figure 3.4: JSON format used when an anomaly is detected

3.3.5 Sink Nodes

Sinks are nodes that subscribe to Redis channels to perform final processing and do not publish their results. They usually serve as interfaces to other systems, such as GUIs, or alarm systems. Examples of sink nodes include the insertion of received data into databases, writing to I/O ports when an anomaly is received, and storing results into files.

3.3.6 Example Anomaly Detectors

Palisade currently includes more than 20 example anomaly detectors. All of these detectors are based on existing methods and are distributed with Palisade to provide out-of-the-box detection of the anomaly symptoms listed in Section 3.2. Tables 3.2 and 3.3 show detectors in Palisade and the anomaly symptoms they are capable of detecting. Table 3.2 shows the detectors for continuous-signal anomaly symptoms (see Section 3.2.2) while Table 3.3 shows the detectors for event-series anomaly symptoms (see Section 3.2.3). In both tables the left-most column shows the detectors while the top-most row shows the anomaly symptoms. The intersection of row and column contains a check mark (✓) if the detector is sensitive to the anomaly symptom and is blank otherwise.

For some detectors, the capability to detect an anomaly symptom depends on the magnitude of the symptom. For example, the spike detector can detect the noise symptom as long as the noise falls outside of the variance of the previous time window. Even though multiple detectors may be able to detect the same anomaly symptom, they complement each other by detecting it in different situations. The distributed nature of Palisade allows running multiple detectors in parallel, increasing the robustness of the system (we show such a situation in the Section 3.5).

Below is a brief description of each of the example detectors. For the more complex detectors, see the relevant citations for a detailed explanation of the algorithm.

Continuous Signal Example Detectors

- **Autoencoders** uses a neural network that can encode and then decode windows of non-anomalous time series [137]. The network is trained by comparing its input to its output and trying to find an encoding that minimizes the difference. Depending on the configuration, the encoding can be on the order of 10 bits for a window of hundreds of samples. When running, the detector encodes and then decodes its input time series using the same network it trained on nominal data. If the network does a poor job of producing output that matches its input (measured by a difference metric), then the detector concludes that the network must not have seen similar data during training, and therefore the input contains symptoms of anomalies.

- **Clip detect** checks for a number of contiguous identically valued symbols at the extremes of the range of a time series. The detector assumes that a sufficiently long sequence of such values is a symptom.

- **SAX + HMM** uses Symbolic Aggregate approXimation (SAX) to discretize a time series into a sequence of symbols [151]. The symbols are based on the distribution of a non-anomalous time series where a new symbol is generated based on thresholds in the learned distribution. The sequence of those symbols are used to build a Hidden Markov Model (HMM) approximating the sequence [228]. The input to the detector is encoded into symbols using the same distributions. Sequences that are sufficiently unlikely in the learned HMM are considered symptoms of anomalies.

- **Sixnum** tracks changes in six standard statistical metrics: mean, standard deviation, maximum, minimum, upper hinge, and lower hinge. Changes to these metrics above configurable thresholds are considered symptoms.

- **Spike detect** continuously computes the variance of the most recent window of a configurable number of samples. A new sample that falls too far outside the variance of the current window is considered a symptom of an anomaly.

- **Spectrum detect** stores a model of the frequency distribution calculated from the Fourier transform of a non-anomalous time series averaged over time. Windows of the input time series are transformed into the frequency domain and compared to the average frequency model. Deviations beyond a configurable distance metric are considered symptoms of anomalies.

Event Series Example Detectors

- **HMM** is similar to the SAX + HMM detector, except that the input is already composed of events that represent state change instead of continuous samples of a signal, so there is no need to transform them using SAX. Since events have a variable time between them, the HMM can consider the time between events, as well as the type of the event when evaluating the likelihood of the sequence against the learned model.

- **Nfer** is a recently introduced language and system for inferring event stream abstractions [129, 130] that utilizes a syntax based on ATL [11]. See Chapter 2 for more details on `nfer`. The integration with Palisade utilizes the `nfer` Python API described in Section 2.5.4. If an interval produced by `nfer` has been designated as anomalous, its generation means a symptom of an anomaly has been detected. Palisade supports both hand-written and mined `nfer` specifications [127].

- **SiPTA** uses the expected periodicity in events from embedded systems to apply signal processing techniques to compare the input traces to non-anomalous data. For more information, see Zedah et al.'s 2014 paper [236].

- **TPG** trains a Task Precedence Graph based on a non-anomalous event stream. This method exploits the periodicity of tasks executed in an embedded system. If the input event stream does not follow the learned graph, it is considered a symptom of an anomaly. For more information, see Iegorov and Fischmeister's 2018 paper [117].

- **TRE** trains Timed Regular Expressions based on a non-anomalous event stream. If an event from the input stream of the detector violates the learned expressions, then it is considered a symptom of an anomaly. For more information, see Narayan et al.'s 2018 paper [172].

Table 3.2: Example detectors and their detected continuous-signal anomaly symptoms

Name	Autoencoders	Clip detect	Range check	SAX + HMM [151, 228]	Sixnum	Spike detect	Spectrum detect
Spike	✓		✓	✓	✓	✓	
S-Wave	✓		✓	✓	✓	✓	
Drifting	✓		✓	✓	✓		
Noise	✓				✓	✓	
Clipping	✓	✓			✓		
Loss	✓	✓	✓	✓	✓	✓	
Smoothing	✓			✓	✓		
Amplification	✓		✓	✓	✓		
Level change	✓		✓		✓	✓	
Frequency change	✓						✓
Echo/Reflection	✓						

Table 3.3: Example detectors and their detected event-series anomaly symptoms

Name	HMM [228]	nfer [129, 130, 127]	SiPTA [236]	TPG [117]	TRE detector [172]
Event Freq. Change	✓	✓	✓	✓	✓
Unexpected Event	✓			✓	
Periods of Silence	✓	✓	✓	✓	✓
Sampled Value		✓			

3.3.7 Fault Handling

Palisade is designed to detect anomalies in streams of data from remote systems, not in its own operation. However, Palisade includes some failure handling capabilities. All nodes in Palisade are monitored by a system service and restarted in the presence of a failure. Detector node failures are interpreted as though an anomaly has been reported by the failed detector, including failures to respond in a configurable time window. This can lead to false-positives, but it is a simple mechanism to alert operators to a situation that deserves their attention. Source node failures are interpreted as loss or period of silence anomaly symptoms by the relevant detectors, and will be reported as anomalies. Sink node failure handling varies depending on the node, but many are obvious (GUI failures) or fail-warn (alarm systems).

3.4 Case Study 1: Autonomous Vehicle

We evaluated the performance and applicability of Palisade using the University of Waterloo's autonomous car as a case study. The vehicle was a 2016 Lincoln MKZ fitted with a range of sensor arrays including LiDAR, a Global Positioning System (GPS) receiver, Inertial Measurement Units (IMUs), cameras, and radars. Figure 3.5 shows an overview of the software and hardware organization in the vehicle. Sensors, such as LiDAR and cameras, produce data that is the input of the autonomy software stack. The output of the autonomy stack (control commands) is sent, for example, to actuators controlling the steering and brakes. Two Renesas automotive computers were installed on the vehicle to run the autonomous driving software. Each computer was equipped with two SoCs with multiple ARM CPU cores and a single ASIL-D certified Micro-controller Unit (MCU).

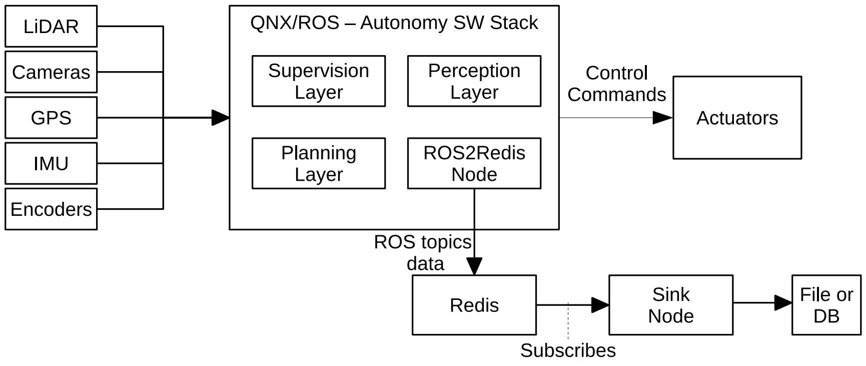


Figure 3.5: Overview of the software/hardware organization in the autonomous car

The autonomous driving software was built using Robot Operating System (ROS). ROS is an open source framework for robotic application development in C++ and Python for POSIX-based Operating Systems (OSs). ROS employs the concept of nodes (a process that performs computation), which provides modularity and development isolation. ROS nodes operate on a periodic loop, are event-driven, or both (they publish data at different frequencies into topics). Like Palisade, ROS uses a publish-subscribe model to communicate between nodes.

We implemented a new ROS node, named ros2redis (see Figure 3.5), to receive messages published to ROS topics and republish them to Redis channels. A Palisade sink node received the command and sensor data and stored them in a database. The stored data were obtained from several ROS topics with different publish frequencies. For instance, GPS information was published at a frequency of 50 Hz, while throttle and gear reports were sent at 20 and 10 Hz, respectively. We logged the data during several autonomous driving sessions and then replayed the recorded data as input to Palisade.

In the next sections, we present the results of running Palisade with the collected data from the autonomous car in two scenarios: gear flip-flop and autonomy mode flip-flop.

3.4.1 Gear Flip-Flop

The autonomous vehicle in the case study reports its current gear in messages published to the `_vehicle_gear_report` topic. The values reported in the data item `_vehicle_gear_report_cmd_gear` reflect the requested shift position of the automatic transmission, and the values in `_vehicle_gear_report_state_gear` reflect the reported shift position. Both values are encoded as integers with the following mapping: {0: No change, 1: Park, 2: Reverse, 3: Neutral, 4: Drive, 5: Low}.

The autonomous driving software sends gear change requests over an Internet Protocol (IP) network to a separate controller that interfaces with the vehicle. The controller then converts the requests into CAN messages that the vehicle transmission understands, and also converts messages from the transmission back into IP packets sent to the driving software. Messages on the `_vehicle_gear_report_cmd_gear` channel are only sent when the software requests a *change*. The request is repeated over a short interval until a message is received on the `_vehicle_gear_report_state_gear` channel reporting that the new gear has been reached. Conversely, the transmission regularly reports its current gear on the `_vehicle_gear_report_state_gear` channel regardless of whether or not it has recently changed.

The reports of the current gear from the transmission exhibit the Sampled Value Anomaly Symptom of Noise. When the value of the gear is stable, the signal appears to be nominal. However, when the gear is changing, the signal varies wildly before finally stabilizing on the correct gear. While it is not clear if intermediate gear values should be reported by the transmission during a gear change, it is clear that the transition should be approximately linear. The fluctuations in the signal amount to noise, and are a symptom that an anomaly has occurred in either the transmission itself or in the CAN controller.

Figure 3.6 shows a brief sample of the two channels over a period when a gear change into *Drive* was requested. The `_vehicle_gear_report_cmd_gear` messages (the blue points) begin at zero, indicating no change is requested, then change to four to request a change to *Drive*, then switch back to *No change* once the gear change is complete. The `_vehicle_gear_report_state_gear` value (the red line) demonstrates the *Noise* Sampled Value Anomaly Symptom as it transitions from *Park* to *Drive*.

We used `nfer` to detect the Gear Flip-Flop anomaly. To highlight the flexibility of Palisade, we implemented two different integrations with `nfer`. The first built Redis and JSON support directly into the C implementation of `nfer`, and the second used a Python processor node to call the `nfer` Python API. The advantage of building support directly in C is its execution speed, while the advantage of calling the tool through its Python API is its simplicity: the Python `nfer` processor is 42 lines of code.

Our `nfer` specification for detecting the Gear Flip-Flop anomaly is given in Figure 3.7. The specification contains one rule which defines conditions which, each time they are met, cause a new interval abstraction to be produced with the associated label (topic), timestamps, and data items also being specified by the rule. The rule says that a new interval abstraction should be published to the `gear_flip_flop` topic when there are

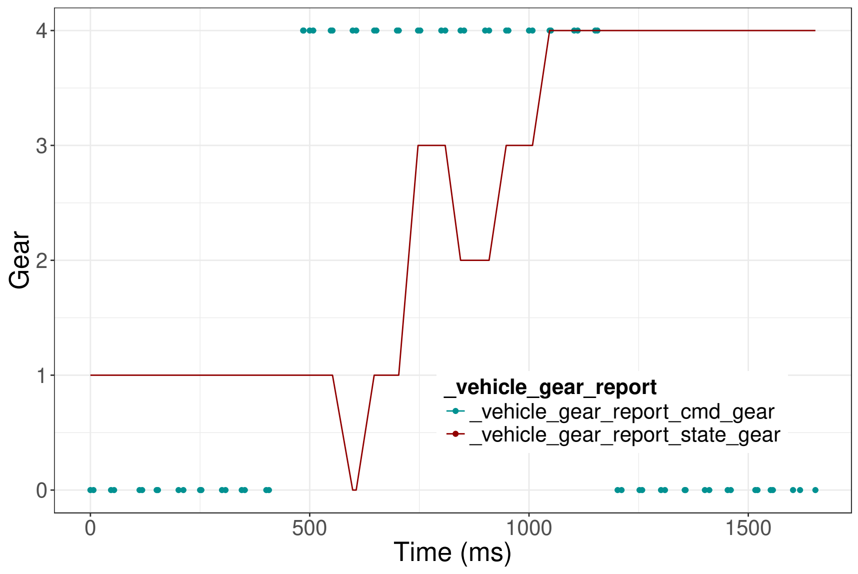


Figure 3.6: Gear flip-flop anomaly example

three messages published to the `_vehicle_gear_report` topic within 200 time units (ms) such that the first and third specify the same gear state, but the second specifies a different gear state.

Line 1 specifies the topic of the resulting message, while Lines 2 and 3 give the topics on which the tool will listen for events. Lines 2 and 3 also partially specify the temporal relationship of those events, and assigns shorthand identifiers (g1, g2, and g3) to each of the events. The **where** clause, from Lines 4 through 9 specifies further conditions that must be met to produce a new interval abstraction. The **map** clause, from Lines 10 through 16 specifies the data items associated with the published abstraction so that consumers of the message can access the details of the anomaly.

We ran `nfer` using our Gear Flip-Flop specification with the *minimality* restriction disabled (--full in the tool) and using the *window* optimization with the window size set to 200. Disabling the minimality restriction causes all matched intervals to be reported instead of the default of only reporting the shortest (minimal) intervals [129]. The window optimization restricts reported intervals to those shorter than the window size, instead of the default of no restriction on interval length [130]. The results are reported in Section 3.4.3.

```
gear_flip_flop :-
  g2:_vehicle_gear_report during
          (g1:_vehicle_gear_report before
           g3:_vehicle_gear_report)
  where
    g1._vehicle_gear_report_state_gear =
          g3._vehicle_gear_report_state_gear &
    g3.begin - g1.end < 200 &
    g1._vehicle_gear_report_state_gear !=
          g2._vehicle_gear_report_state_gear
  map {
     first _gear -> g1._vehicle_gear_report_state_gear,
    anomalous_gear -> g2._vehicle_gear_report_state_gear,
    anomaly -> 1,
    note -> "The gear changed and reverted within 200 ms",
    channel -> "_vehicle_gear_report"
  }
```

Figure 3.7: nfer specification for detection of Gear Flip-Flops

3.4.2 Autonomy Mode Flip-Flop

The autonomous vehicle reports its current autonomy state in messages published to specific topics, such as `_vehicle_brake_report` and `_vehicle_throttle_report`. When the data item `enabled` is true, the car is running in autonomy mode. When `enabled` is false, the vehicle is controlled by the human driver.

Once enabled, autonomy mode should remain enabled, as indicated by the `enabled` data item, for the duration of a trip. However, in track testing, we found instances when the vehicle would switch from autonomy enabled back to autonomy disabled during a lap, giving the driver control of the vehicle. We discovered that the Dataspeed CAN module of the autonomous vehicle would disable autonomy mode if no command was given to the vehicle for at least 80 ms. This timeout was unexpected and caused an unsafe driving condition. When the timeout occurs, and autonomy mode is disabled, we call the condition Autonomy Mode Flip-Flop.

We used TREs to monitor the Autonomy Mode Flip-Flop anomaly. Regular expressions provide a declarative way to express patterns for a system specification. TREs define timing constraints in a regular expression [172]. For instance, a regular expression specification of the form “state a is followed by state b” can be modified to “state a is followed by state b within t time units” to obtain a TRE specification. The TRE processor uses a manual specification of a TRE to monitor and detect anomalies. To integrate with Palisade, we added Redis support to the C implementation of the TRE detector.

The TRE specification used for detecting the Autonomy Mode Flip Flop anomaly is the following

$$((\langle P.S\rangle\,[0,3])||(\langle S.P\rangle\,[0,3]))+$$

P and S above represents the two Autonomy Mode Flip-Flop states - Autonomy Enabled and Disabled respectively. The above specification can be translated as the occurrence of patterns where the flip-flop changes from enabled state to disabled state or vice versa within 3 time units.

3.4.3 Comparison with Siddhi

To evaluate Palisade's rules-based capabilities, we matched the `nfer` processor against the Complex Event Processing (CEP) system, Siddhi. We developed a query in the Siddhi query language to find all instances of the gear flip-flop anomaly described in Section 3.4.1, then modified the `nfer` specification in Figure 3.7 to exactly match the results from Siddhi. By first configuring Siddhi, we were able to keep its query short and natural, while the specification for `nfer` to exactly match its result was more complex. The purpose of this choice was to avoid biasing the test results against Siddhi. The data and configuration used for this comparison is available at [126]. In this test, Palisade detected anomalies over 35 times faster than Siddhi and with much lower variance in the latency.

Siddhi is a good choice to compare with the `nfer` processor of Palisade because it is a specification-based stream-processing framework. Both Siddhi and `nfer` require the user to write rules in a declarative language to generate facts from a stream of input events. Both languages are complex enough to define queries for the Gear Flip-Flop anomaly: `nfer` is Turing complete when circular references are permitted [130] and Siddhi is likely Turing complete, although no complexity analysis is available [217]. Like Palisade, Siddhi supports data streaming frameworks where sources and sinks may be remote from the processor. While Siddhi supports cloud-based installations, it can also be installed locally and deployed where internet connections are not available. The ability to install the software locally was important both for automotive use cases and for our ability to accurately measure the tool's detection latency. Siddhi is also easy to install, which is a requirement for Palisade that many CEP systems fail to meet.

We used Siddhi and the `nfer` Palisade processor to independently monitor events sent over a network. We ran the test on a desktop computer with an Intel I5-5200U 2.7 GHz processor and 8 GB of memory running Linux 3.10.0. For Palisade, we streamed the data over Redis and for Siddhi, as it did not support Redis, we sent the data directly over HTTP. For the data, we chose a period of about 68 minutes during which 73,079 events occurred. To simulate an online environment, we delayed publication between messages for the same period as the difference in their timestamps. For example, the difference between the timestamps of sequential messages A and B was 75 ms, we would publish message A and then delay 75 ms before publishing B.

Figure 3.8 contains part of the Siddhi query to detect the Gear Flip-Flop anomaly. In the query, Lines 1-4 match the input stream when three events occur within 200 ms where the first and third event report gear zero but the second reports a different gear. If there is a match, the code in Lines 5-12 are a select statement for the data to be published, while Line 13 sends the data on the output stream over HTTP where we record the capture. The omitted portions of the query repeat Lines 1-13, but replace the gear value of zero in the first two lines with the other possible gears. For more information on the Siddhi query language, see the Siddhi documentation [204] and previous publication [217].

```
1   from every e1=dataInputStream[gear == 0],
2              e2=dataInputStream[gear != 0],
3              dataInputStream[gear!=e1.gear]*,
4              e3=dataInputStream[timestamp - e1.timestamp < 200 and gear
                   ==e1.gear and timestamp > e1.timestamp]
5   select     e1.timestamp as Timea,
6              e1.gear as Geara,
7              e2.timestamp as Timeb,
8              e2.gear as Gearb,
9              e3.timestamp as Timec,
10             e3.gear as Gearc,
11             e3.pySendTime as pySendTime,
12             eventTimestamp() as siddhiSendTimestamp
13  insert into outputDataStream ;
```

Figure 3.8: Partial Siddhi specification for detection of Gear Flip-Flops

To measure round-trip detection latency, we added the time of publication to each event and then copied those values to the output when a gear flip-flop anomaly was detected. A second script monitored the results and recorded the timestamp when the anomaly report was received. We then subtracted the passed-through publish timestamp of the message that triggered the anomaly report (the most recent message) from the timestamp when the anomaly report was received.

The comparison in Table 3.4 shows that, in this case, Palisade detects anomalies over 35 times faster than Siddhi. In the table, lower lower values are better for both the mean and stardard deviation of the latency. Not only is Palisade's mean detection latency much faster, but the standard deviation of its latency is also about 2.4% of that of Siddhi.

Table 3.4: Results of comparing Palisade with Siddhi

Round-trip latency	**Siddhi**	**Palisade**
Mean (lower is better)	60.6 ms	1.58 ms
Standard deviation	20.9 ms	0.50 ms

3.5 Case Study 2: ADAS-on-a-Treadmill

Advanced Driver-Assistance Systems (ADAS)-on-a-Treadmill is a research platform of the Real-time Embedded Software Group of the University of Waterloo [203]. The platform mimics the movement of car on a straight road using a treadmill. A model car with on-board ADAS features emulates various driving conditions, such as Adaptive Cruise Control (ACC), Lane Keeping Assistance (LKA), Lane Departure Warning (LDW), and Forward Collision Detection and Avoidance (FCDA). As in the autonomous vehicle case study, ADAS algorithms are implemented on top of ROS. We have integrated Palisade with the ADAS-on-a-Treadmill platform and run four anomaly detection algorithms (spike, clipping, loss, and range) in two different scenarios (GPS spoof and dead spot). The next sections describe each scenario and present an evaluation.

3.5.1 GPS Spoof Attack

The GPS spoof scenario simulates an attack on the vehicle positioning system. In this scenario, one car moves from one side of the treadmill to the other on the Y-axis, while keeping the same position on the X-axis. The GPS spoof attack changes the car's Y position by fooling the controller into correcting for an inaccurate reading. In a real autonomous vehicle, such an attack could result in an accident, for instance. We performed three

GPS spoof attacks in a five minute period and ran two Palisade detector nodes, spike and clipping.

The spike detector keeps each data point in a buffer. Whenever the buffer is full and new data is received, the detector discards the oldest data point. Then, it compares the value of the newly received data with the mean and standard deviation (std) of the data in the buffer. If the received value is greater or smaller then the std multiplied by a constant plus the mean, then an anomaly is reported. Once an anomaly is detected, the detector waits for a period before starting to compute the mean again. The buffer length considered was 50, the std multiplicative constant was 4.6, and the waiting period was 8 seconds. These are all configurable parameters in the detector. We chose these parameters because they output anomalies without false positives (we discuss the choice of detectors parameters in Section 3.6).

The clip detector also buffers incoming messages to avoid detection jitter. When the clip detector fills its buffer, it counts how many data points in the buffer are within a configured distance of new values (buffer value + distance > received value *and* buffer value - distance < received value). When there are ten or more matching data points within the interval, a clipping anomaly is reported. The buffer length used in the experiment was 60 and the distance parameter was 0.0005 (difference in the received Y-axis position).

Figure 3.9 shows the car's Y position when the attacks were performed and the output from the two detectors. The spike detector identifies two out of the three anomalies, while the clip detector finds all three inserted anomalies. The first anomaly is identified quickly by the spike detector because there is a sharp jump in the received Y position. We also ran both detectors with the same parameters using a dataset without anomalies, and neither detected any anomalies, as expected.

3.5.2 Dead Spot in a Platooning Formation

This scenario simulates a situation where a lighting inconsistency in the environment causes "dead spots" on the conveyor, causing any car that drives into the spot to lose its positioning data. This is inspired by a real experience running the University of Waterloo autonomous car at CES 2018. In this scenario, two cars drive on the treadmill in a platoon formation. A dead spot is inserted in the treadmill and a command is issued to move the first car forward on the X-axis. The car then moves to the desired point, stays for a while, and returns to its original position. We ran the experiment for five minutes and inserted four dead spots while executing two Palisade detector nodes, loss and range check.

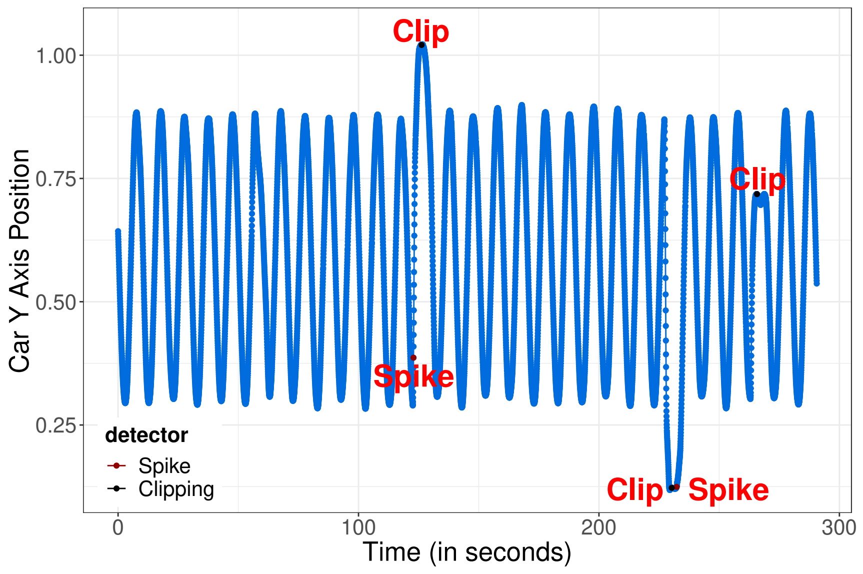
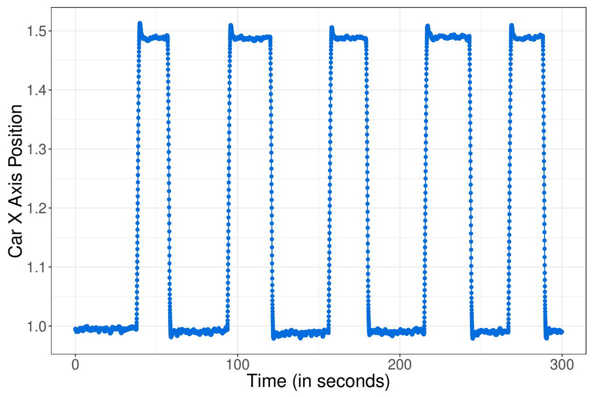


Figure 3.9: Car positioning with four inserted anomalies and the anomaly detection points (Spike and Clipping detector)

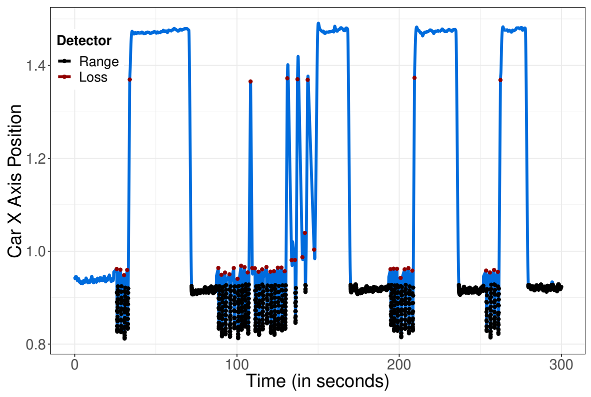
The loss detector keeps track of the average period of message reception and, when a data point takes longer to arrive than the average multiplied by a constant factor, an anomaly is reported. The detector checks for anomalies after a minimum number of samples is received. In the experiment, we set the minimum number of samples to 30 and the constant factor to four. The range detector tests whether a data point is between a maximum and minimum value. If a received data point is outside the range, then an anomaly is reported. We trained the range check with a dataset without anomalies.

Figure 3.10a illustrates the expected behavior (without anomalies) of the platooning formation for the first car. For instance, around 100 seconds into the experiment a command to move the first car forward is issued, which causes it to move from the position of about 1.0 to 1.5. When the vehicle reaches the desired position, it takes a second or two to stabilize, causing the small spikes after each acceleration.

Figure 3.10b shows the car X-axis position with inserted dead spots. The first command to move forward is issued around 25 seconds into the experiment. The first car attempts to move to the desired point but reaches a dead spot where it loses its positioning signal for a short time. This causes the “shaking” at the bottom of the figure as the controller tries to reestablish the car’s position. The range detector is able to identify such a situation because those values are lower than the minimum in the dataset without anomalies. The



(a) Nominal car positioning



(b) Car positioning with dead spots

Figure 3.10: Car positioning with inserted dead spots and the anomaly detection points (Loss and Range detector)

loss detector recognizes the loss of communication while the car passes through a dead spot. Around 140 seconds into the experiment, we can see two lines that move up and down in a short period. This happens because the first car passes the dead spot by its side, corrects its trajectory by returning to the dead spot, and then returns to its position before the command to move was issued. This is another scenario where Palisade improves the anomaly detection by providing means to easily run two detectors using the same input data. We discuss how Palisade improves the anomaly detection in Section 3.6.

3.5.3 Comparison with Beep Beep 3

We evaluated Palisade against the stream-processing system Beep Beep 3. We developed a Beep Beep 3 processor to find all instances of the dead-spot anomaly described in Section 3.5.2. To provide an accurate comparison, we used the Beep Beep 3 HTTP palette to construct a distributed detector, with events sent and anomalies reported over a network connection. We found that Palisade and Beep Beep 3 attained comparable anomaly detection latency in this case, but that building such a distributed processor with Beep Beep 3 was more cumbersome than with Palisade. The data and configuration used for this comparison is available at [126].

Beep Beep 3 is a good choice for a comparison with Palisade because it is specifically designed for online, streaming data processing. Futhermore, the tool is highly flexible, supports arbitrary data types, and allows distributed processors to be created using official libraries. Like Palisade, Beep Beep 3 is more of an architecture and set of APIs than a standalone tool. However, unlike Palisade, Beep Beep 3 does not include out-of-the-box processors designed for anomaly detection, although such extensions exist [197].

We used Beep Beep 3 along with the Palisade RangeCheck and LossDetect processors to independently monitor events sent over a network. We ran all the detectors on a desktop computer with an Intel I5-6300U 2.4 GHz processor and 16 GB of memory running Linux 4.19.72. Palisade was run on Python 3.6.10 and Beep Beep 3 was executed using OpenJDK 1.8.0_252 (IcedTea 3.16.0). For Palisade, we streamed the data over Redis and for Beep Beep 3, as it did not support Redis, we used the Beep Beep 3 HTTP Palette and its serialization library. For the data, we used the same period of about 5 minutes from Figure 3.10b during which the car's position was reported 7,030 times. To simulate an online environment, we delayed publication between messages for the same period as the difference in their timestamps. For example, the difference between the timestamps of sequential messages A and B was 33 ms, we would publish message A and then delay 33 ms before publishing B.

To measure round-trip detection latency, we added the time of publication to each event and then copied those values to the output when a loss or range anomaly was reported. A second program monitored the results and recorded the timestamp when the anomaly report was received. We then subtracted the passed-through publish timestamp of the message that triggered the anomaly report (the most recent message) from the timestamp when the anomaly report was received.

To compare between Palisade and Beep Beep 3, we constructed a Beep Beep 3 stream processor that mimicked the behavior of both the RangeCheck and LossDetect Palisade processors. Figure 3.11 shows an outline of this processor, along with Beep Beep 3 programs for reading and printing events [103].

Figure 3.11 uses the official Beep Beep 3 drawing guide to show how events are read, transmitted, filtered, retransmitted, and printed. The top diagram in the figure shows the events being read from a file and transmitted using the HTTP palette to the central processor. The central diagram in the figure shows how events are filtered to only be included in the output if they are out-of-range or occur after a long delay. The events arrive via HTTP and are duplicated to follow two paths: in the lower path, they are tested to see if they are out-of-range or occur after a long delay (there is more logic here, not shown in the figure), in the upper path, they reach a filter (shown as a traffic light) which is gated based on the result of the lower path test. Events for which the test is true are transmitted via HTTP to the final program, shown as the third diagram. The third diagram shows how events that arrive are then printed to the console.

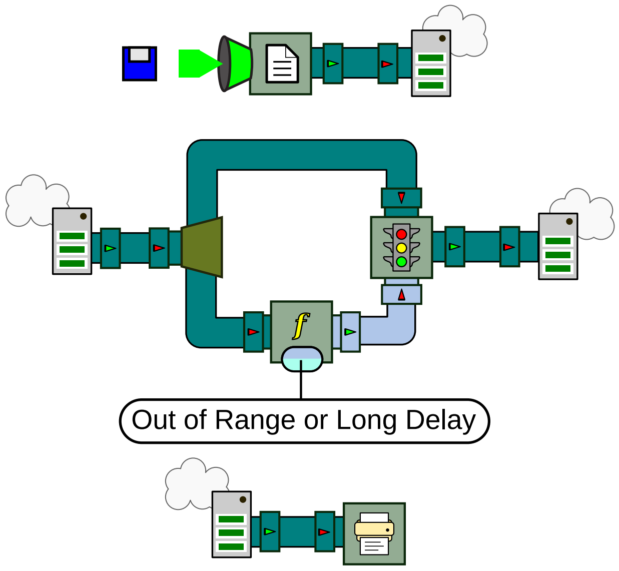


Figure 3.11: Graphical representation of the Beep Beep 3 processor omitting the details of how Range and Loss were computed

Beep Beep 3 is not designed for distributed processing, however. The only officially supported networking mechanism is direct HTTP connections, which necessitates tightly coupled components. That is, the program that reads the events from a file must know the address of the processor, which must, in turn, know the address of the program that prints the events. This is why the range and loss checks are performed in one processor; if they were separated into two processors, the file reader would need to send events directly to both end points. That Palisade components are loosely coupled is a fundamental advantage for distributed stream processing.

Although it was not necessary to write new Palisade processors for the comparison, the existing processors are much simpler than the equivalent processor written with Beep Beep 3. The reason for this disparity is that the programming model for Beep Beep 3 is not aligned with the programming language in which it must be implemented. That is, Beep Beep 3 programs do not resemble Java programs. This can most clearly be seen in the code to apply a function to check if an event is outside a fixed range. In Palisade, this check is written in Python and requires a series of three conditional statements with inequality expressions. In Beep Beep 3, the check requires the creation of 14 objects, essentially requiring the author to create an abstract syntax tree by hand. Beep Beep 3 does support the creation of DSLs but intentionally avoids providing one [1].

The comparison in Table 3.5 shows that, in this case, Palisade and Beep Beep 3 had similar mean detection latency but the standard deviation of Palisade's latency was lower (lower is better). The higher standard deviation for Beep Beep 3's detection latency appears to be the result of the Java Virtual Machine's Just-In-Time (JIT) compiler as a few anomalies early in the experiment had many times longer detection latencies than the rest. These longer detection times could probably be avoided by implementing a boot-strapping process for the Beep Beep 3 processor, but this would require another component and added complexity not required by Palisade.

Table 3.5: Results of comparing Palisade with Beep Beep 3

Round-trip latency	**Beep Beep 3**	**Palisade**
Mean (lower is better)	2.87 ms	2.85 ms
Standard deviation	3.2 ms	0.66 ms

3.6 Discussion

This section discusses the Palisade results and design choices. We divide the discussion in three parts: software architecture, performance, and anomaly detection.

3.6.1 Software Architecture Evaluation

Evaluating software architectures is not a straightforward task. There is no common language to describe different software architectures and no clear way to understand and compare different software concerns, such as maintainability, portability, modularity, and reusability [133]. Also, the effectiveness of the software architecture is related to the experience and knowledge of the development team, thus quality must be considered in this context.

We examined two software architecture evaluation methods, Software Architecture Analysis Method (SAAM) [133] and Architecture Tradeoff Analysis Method (ATAM) [134], to determine if we could objectively evaluate Palisade's architecture. We found that both

[1]The API provided by Beep Beep 3 could arguably be described as a deep internal DSL

methods are intended for evaluating monolithic software projects that serve specific business cases, and that they do not map well onto Palisade. However, both SAAM and ATAM argue that *modularity* and *extensibility* are important metrics for evaluating the quality of an architecture. While we cannot quantitatively measure these metrics, here we present an argument that they are well satisfied by our design.

One measurement of the extensibility of software is the number of lines of code that must be written to meaningfully extend it. The most common way to extend Palisade is to write a new processor node. The average number of lines in the current Palisade processor nodes (written in Python) is 144.25 lines (including comments). Autoencoder is the longest processor node with 601, while Clip detector is the shortest with 72 lines. The *Processor* abstract base class has 239 lines. The `nfer` detector, which uses the `nfer` Python API, has only 42 lines of Python code.

As discussed in Section 3.3.4, processor nodes are independent modules that share infrastructure from a base class. Editing a processor node has no effect on upstream or sibling processor nodes. Only nodes dependent on the output of the edited node may themselves require editing.

Constructing a new source node does not affect other source nodes in the system. Only processor nodes that will be subscribing to a new source may need adjustment, and then only if the new source differs from those that already exist. Adding a new processor node has a lesser impact than editing one, as no downstream nodes should be affected, typically. Instead a new processor node can be added to the system without a single modification to any other component.

For adding a new processor node, we consider the basic code to extend the base class. A new processor node requires at least 24 lines of code in Python. Obviously, the total number of lines depends on the complexity of the algorithm, but the processor abstract base class makes extending Palisade straightforward.

We compared the extensibility of Palisade with the CEP/RV system Beep Beep 3 In Section 3.5.3. While it was possible to build processors in Beep Beep 3 that mimicked those in Palisade, they required tight coupling between components. In Beep Beep 3, constructing a new processor or sink node would require modifying the other nodes. Palisade's loose coupling between components means that these similar modifications are not required to support new or modified nodes.

Palisade can be used in any embedded system that provides a network interface. As the core Palisade functionality is built around the Redis publish-subscribe interface, any system that has a network interface can send data directly to Redis or to a server that then sends to Redis. Also, RESP is simple and would be easy to port to an embedded system without Linux support. Consequently, we believe that the integration of Palisade with any embedded system is a straightforward task.

3.6.2 Performance Evaluation

Palisade is built for low latency anomaly detection and this is evident from our comparisons with other frameworks. In the case study evaluation in Section 3.4.3, an `nfer` Palisade processor detected anomalies over 35 times faster than a comparable detector using the CEP system Siddhi. In the case study evaluation in Section 3.5.3, two Palisade processors detected anomalies with similar latency to a comparable detector using Beep Beep 3, which required tight coupling between components and a using a complicated API for performing simple data comparisons.

We looked for other appropriate frameworks to compare against Palisade detectors such as the Autoencoder processor, but we discovered that no such framework exists. It does not make sense to compare Palisade's performance against a framework which does not support many of the same core features or which is unusable in the same environments. Frameworks such as Extendible and Generic Anomaly Detection System (EGADS) [82] and Datastream.io [61] only support CSV input for offline detection, while Palisade operates online. Other frameworks like Esper [221] and TeSSLa [54] support online stream processing, but lack support for distributing processors over a network. Detectors like Hogzilla [116], StreamMill [222], and NiagaraCQ [48] are abandoned projects that cannot be installed. Others, like Thirdeye [60], can only be run in a cloud environment, making them ill-suited for latency comparisons. The lack of online, streaming, distributed, locally runnable anomaly detection frameworks shows the need for Palisade, and we hope that our work motivates others to design comparable tools.

An important design decision in Palisade regards the copying of messages instead of passing message IDs. Once data arrives into a channel, Redis copies the messages to all nodes that subscribed to that specific channel. Another approach, found in Zero-Copy message protocols [218] for example, would be to pass just the message ID to all destination nodes. The ID would then be used to access a central database to retrieve the data. When most nodes require the data, however, the ID passing approach causes a performance bottleneck due to access serialization at the central database (increased latency). We assume that a node that subscribes to a channel needs the data on that channel, so the message copying approach reduces latency while not affecting the processing or memory requirements. This is a reasonable assumption given that it only requires different types of data to be assigned separate channels. The ID passing approach is usually used in micro-service architectures and is preferable when the target application needs all of the data (good for batch processing) [74].

3.6.3 Anomaly Detection

The multiple anomalies detected by different processors can be compared against each other to verify anomalies and thereby decrease the false positive rate of anomaly detection by Palisade (this could be done by a voter sink node, for instance). There are also cases where some anomalies are detected by only a subset of the detectors. Palisade covers these cases as a variety of detectors can be integrated with low-development effort (due to our design choices - command channel, abstract base class, and data formats).

The choice of parameters in the detector nodes plays a central role in the efficiency of such detectors. In our experiments, we varied the detector parameters until we found a configuration without false positives (Sections 3.4 and 3.5). This was possible because we could repeat the execution of the detectors several times. When the execution cannot be repeated, we suggest tuning the detector parameters using a system simulation.

3.7 Related Work

This section describes existing work related to Palisade. We divide the discussion by subject area: anomaly detection, Information Flow Processing (IFP) systems, anomaly detection with streaming frameworks, offline frameworks, and outdated or commercial frameworks. Few existing works combine the central features of Palisade: online, distributed anomaly detection for both time series and event streams. Our work is motivated by the lack of options in this niche area.

3.7.1 Anomaly Detection

Anomaly detection, sometimes called outlier detection, attempts to find unexpected or non-conforming patterns in data [8, 44, 176, 19, 9, 184]. Anomalies are distinct from noise, in that noise is not of interest and hinders analysis. The output of an anomaly detector may be either a *score* or a *label*, but the purpose is always to provide a verdict on whether an anomaly was detected at a given time.

Anomaly detection has appeared in statistics literature for many decades [80, 183], but more recently it has found application and been studied in other fields. In healthcare, anomaly detection is used to look for cardiac irregularities that might indicate heart failure or patterns of disease outbreak [98, 231]. In computer network security, anomaly detection is widely used in intrusion detection systems to look for suspicious activity [228, 209, 5]. Banks, insurance companies, and advertising firms, among others, employ anomaly detection to search for instances of fraud [87, 72, 97]. Heavy industry and safety-critical systems operators like airlines use anomaly detection for equipment damage detection [136, 29]. Recent work has shown how anomaly detection can be applied to detect events in a power grid [239].

Lightweight Online Detector of Anomalies (LODA) is a data streaming online anomaly detection system [182]. LODA uses a collection of one-dimensional histograms to improve the anomaly detection. The rationale behind the use of a collection of weak classifier is because together they can form a strong classifier [139]. LODA presented the same performance in terms of precision of HS-Tress, but with better time to process a data stream.

Weber et al. proposed a two-stage anomaly detection framework for vehicle signals [229]. The first stage is based on static checkers (for CAN messages) and the second stage is based on machine learning algorithms (named learning checks). The learning checks stage implements a Recurrent Neural Network (RNN) and LODA. In a performance evaluation using CAN messages from a vehicle, RNN had a false positive rate of 0.065%. Palisade also supports both static and machine learning-based algorithms. However, Palisade supports the execution of all detectors in parallel or in any number of stages (not only two). In the same work, the authors defined seven types of anomalies that can occur in a sensor or Electronic Control Unit (ECU) [229]. Table 3.6 compares the seven types of anomalies propose in [229] with our nomenclature described in Section 3.2. We can note that the seven types of anomalies are a subset of ours. In this sense, we provide a more comprehensive overview of anomaly symptoms that can occur in embedded real-time systems.

Table 3.6: Anomaly symptoms defined in [229] compared to our proposed symptoms

Symptoms in [229]	**Our Symptoms**
Sine anomaly	S-Wave
Plateau stuck anomaly	Loss
Peak anomaly	Spike
Negative peak anomaly	Spike
Noise	Noise
Plateau rise/fall anomaly	Clipping
Zero fall anomaly	Clipping/Loss

3.7.2 Information Flow Processing Systems

Palisade processes flows of information online, deriving high level events and alerts from the flow as data is received. This online flow processing has similarities to the definition by Cugola and Margara of a CEP system [56]. CEP systems are a kind of IFP system that supports continuous and timely processing of low-level event streams into high-level abstractions according to predefined rules. CEP systems are differentiated from Data Stream Management System (DSMS) systems in that DSMS systems work on generic data streams rather than event streams and their output is similarly unconstrained. Palisade supports both event and non-event stream information and includes processors, like `nfer`, that use predefined rules, and learning-based processors that use training data to construct behavioral models.

This section discusses some CEP systems in the literature that could be applied to some of the use cases for Palisade. However, all of these systems require predefined rules to construct their event abstractions, which most Palisade processors do not. As a result, these CEP systems should only be compared to Palisade processors with the same requirements.

Gigascope is a DSMS designed for network monitoring, intrusion detection, and traffic analysis [55]. Gigascope uses a Structured Query Language (SQL)-like query language called Gigascope Query Language (GSQL) that uses data streams as its input and output. Gigascope is explicitly aimed at intrusion detection in networked systems and is not a general solution for anomaly detection.

Triceps is an open source CEP system that does not define its own SQL variant, but rather has the user implement queries and operations directly in C++ or Perl [1]. Triceps is unique in that it is an embedded CEP. That is, Triceps is meant to be used as a library and to be embedded into other programs. This fills an interesting niche, but it is not a framework for distributed anomaly detection like Palisade. In the future it would be interesting to build a processor using Triceps, similar to the already existing `nfer` processor.

Esper is a CEP and DSMS for Java and .Net (Nesper) with a SQL variant called Event Processing Language (EPL) that Esper compiles to byte code [221]. Esper is designed for low latency and high throughput, as well as extensibility and low resource utilization. These traits make Esper a good candidate for online anomaly detection. Esper is designed to work well running inside a distributed stream processing framework and includes examples of integrations with Java networking libraries. However, Esper is not itself a distributed stream processing framework, and integrating Esper with Palisade would require building a variety of custom components to handle networking, serialization, and command.

Siddhi is an open source CEP system deployed by companies such as Uber, eBay, and PayPal for use cases like fraud analysis and policy enforcement [217]. Siddhi uses a specification language called Streaming SQL and supports input from a variety of streaming sources such as Apache Kafka and NATS in diverse formats like JSON and Extensible Markup Language (XML). It supports streaming input and ouput, multiple end-points, and specification-based anomaly detection, and is one of the existing works closest to supporting Palisade's requirements (defined in Section 3.3). We compared the detection latency of Palisade with Siddhi, where detection latency is defined as the time difference between the instant data is generated by a source and the instant it is reported as anomalous by a detector. The results, reported in Section 3.4.3, show that Palisade responds over 35 times faster on average than Siddhi for our case study.

3.7.3 Anomaly Detection with Streaming Frameworks

Several other works have used data streaming frameworks for online detection of errors or anomalies. Lopez et al. discuss the characteristics and compare the throughput of three stream processing platforms (Apache Spark, Flink, and Storm) using a threat detection application [155]. Solaimani et al. used Apache Spark to detect anomalies for a multi-source VMware-based cloud data center [210]. Subramaniam et al. proposed a framework to detect anomalies online (outlier detection) in wireless sensor networks [216]. However, the authors only implemented the framework in a simulator. Du et al. built a streaming detector using Apache Storm that used k-Nearest Neighbors (k-NN) to detect anomalies in IP network traffic [76]. Shi et al. implemented an online fault diagnosis system based on Apache Spark for power grid equipment [202]. Song et al. proposed an integrated system for explainable anomaly detection using Apache Spark called EXAD [211].

Thirdeye is an anomaly detection framework based on Apache Spark [60]. It uses machine learning and artificial intelligence algorithms for cybersecurity, data analytics, and outlier detection. Thirdeye is designed for deployment on Amazon's AWS cloud computing platform. Palisade, by contrast, is designed to run on a curated local area network to reduce communication latency.

Beep Beep 3 is a stream-processing system that combines some aspects of CEP systems with ideas from RV [102]. Beep Beep 3 is primarily a set of Java APIs to build synchronous processors for arbitrary data types. The standard Beep Beep 3 APIs may be augmented with modules called *palettes* that implement interfaces such as network communication, temporal logic, and plots.

Beep Beep 3 has been studied for use in online, streaming anomaly detection [197]. However, Beep Beep 3 has different goals from Palisade that make it less well suited for distributed anomaly detection. We compared the detection latency of Palisade with Beep Beep 3, as well as the experience of building anomaly detectors using the two frameworks, in Section 3.5.3. The results show that the two tools have similar detection latency, but that Palisade is better suited than Beep Beep 3 for detecting anomalies in a distributed environment.

TeSSLa is a stream-based specification language and monitoring system designed for specifying and analyzing the behavior of systems where timing is important [54]. TeSSLa, like Beep Beep 3, combines aspects of CEP systems with RV. However, unlike Beep Beep 3, TeSSLa may only be used through an external DSL and its interpreter only accepts input via file arguments or standard in. While TeSSLa's language and theoretical foundation are exciting, its lack of network support means that it cannot currently operate as a distributed, online anomaly detection framework. Integrating TeSSLa with Palisade is also impossible because TeSSLa is only distributed as a compiled binary.

3.7.4 Offline Frameworks

Datastream.io is an open-source anomaly detection framework that allows users to integrate their custom detectors for testing and training [61]. The project plans to support online streaming but presently only supports the use of CSV files as input to perform offline detection.

EGADS is an open-source anomaly detection framework by Yahoo [82]. EGADS is a self-contained Java package developed for time-series anomaly detection, providing access to multiple detectors. EGADS accepts input only in the form of CSV and standard-input and is no longer actively maintained.

Frankowski et al. used a variety of CEP systems, including Siddhi and their own SECOR CEP, to detect intrusions and anomalies [94]. Their work combined several CEP systems to periodically analyze log files and store the results in a database. They showed that it is possible to build an effective, signature-free anomaly detection framework using off-the-shelf components. However, they did not construct an online detector.

Outdated or Commercial Frameworks

Other frameworks have been proposed in the past but are unusable in practice because they are either expensive to license or unmaintained. NiagaraCQ was an early and influential continuous query system, but the software has not been available for at least a decade [48]. SASE was a stream processing system designed to support complex queries and high throughput, but it was last maintained in 2014 [238]. Cayuga was a stateful publish-subscribe system based on Non-deterministic Finite Automata (NFAs) that was adapted as an event monitoring system, but it was last maintained in 2013 [65]. Stream Mill was a DSMS that combines predefined rules with statistical learning algorithms for mining queries [222], but it was last maintained in 2012. GEM is a commercial CEP vendor in the industrial space and, as such, their software is not freely available [96].

Hogzilla is an open-source anomaly-based Intrusion Detection System (IDS) targeted towards network communications [116]. Hogzilla purports to detect a wide range of network attacks including zero-day attacks. At the time of publishing, the software no longer runs and has not been maintained for some time. However, in October 2019 the project website was updated to report that the tool will be maintained and supported by a commercial partner, so Hogzilla may return to relevance in the area.

Many of the IFP systems Cugola and Margara review are either no longer maintained or locked up behind commercial licenses [56]. Other promising systems from their study that are unavailable or impractical include: the Borealis stream processor (abandoned 2008), StreamBase (commercial), SQLStream (commercial), Oracle CEP (commercial), Tribeca (disappeared), and TelegraphCQ (abandoned 2003).

3.8 Conclusion

In this chapter, we presented Palisade, a software framework for anomaly detection in embedded systems. We introduced a new taxonomy of anomaly symptoms, and we designed Palisade to support their detection using a variety of algorithms including `nfer`. Palisade is built around the Redis publish-subscribe interface, which allows running different anomaly detectors with the same input data across a distributed network. We demonstrated the viability of the proposed framework using two case studies, one using data from an autonomous vehicle and another one using data from an ADAS platform. We argued that Palisade is easy to operate and modify and that it detects anomalies with low latency.

As future work, we plan to integrate Palisade with the University of Waterloo's autonomous car and implement more learning-based anomaly detectors. The datasets used in the experiments is available online [126] and the Palisade source code is available upon request.

Chapter 4

Monitoring Over Unreliable Channels

4.1 Introduction

In RV the correctness of a program execution is determined by another program, called a monitor. In some cases, monitors run remotely from the systems they monitor, either due to resource constraints or for dependability. For example, ground stations monitor a spacecraft, while an automotive computer may monitor emissions control equipment. In both cases, the program being monitored must transmit data to a remote monitor.

Communication between the program and monitor may not always be reliable, however, leading to incorrect or incomplete results. For example, data from the MSL rover is received out-of-order, and some low priority messages may arrive days after being sent [81]. Function tracing in Linux with the *ftrace* framework requires careful filtering to avoid losing as many as half of all kernel events from full message buffers [213]. Even dedicated debugging channels like ARM Embedded Trace Macrocell (ETM) have finite bandwidth and may lose data during an event burst [15]. Some works in the field of RV have begun to address the challenges of imperfect communication, but the problem has been largely ignored in the study of monitorability.

This chapter introduces a definition for a property to be considered monitorable over an unreliable channel. We define common mutations that may occur to a trace and provide a decision procedure to test ω-regular properties for monitorability over a channel with such a mutation. We also definine when a property can be unmonitorable over an unreliable channel but still have value to monitor. We also provide a classification of properties that may be monitored over certain unreliable channels.

The chapter is organized as follows. We first define notation used throughout the chapter in Section 4.2. We then introduce foundations necessary for understanding the chapter in Section 4.3, first examining the concept of uncertainty in monitoring in Section 4.3.1 and then reviewing common notions of monitorability in Section 4.3.2. We then define common trace mutations due to unreliable channels in Section 4.4. In Section 4.5, we describe what makes a property immune to a trace mutation and how that relates to monitorability. Section 4.6 expands on that idea by defining how a verdict for a property may be trustworthy over an unreliable channel even when the property is not immune to the channel's mutation. We then review and augment the Finitely-Refutable/Finitely-Satisfiable property classification in Section 4.7 by adding subclasses relevant to common mutations. We use this augmented classification to categorize properties with trustworthy verdicts over those mutations in Section 4.8 including a discussion of the utility of such properties in Section 4.8.5. We work towards a decision procedure for the immunity of an ω-regular property by mapping the definition of immunity to a property of derived monitor automata in Section 4.9. Finally, we present a decision procedure for the immunity of an automaton to a mutation and prove it correct in Section 4.10. We then present related work in Section 4.11. Section 4.12 discusses some of the conclusions from the chapter and possible future work.

4.2 Preliminary Notation

This chapter uses propositional symbols instead of events with timestamps and data. This choice limits the expressive power of traces and is important for many of our results. AP is a finite, non-empty set of *atomic propositions*. Throughout the chapter, we assume an *alphabet*, denoted $\Sigma = 2^{\text{AP}}$. An element of the alphabet is a symbol $s \in \Sigma$. In this chapter, a *trace*, *word*, or *string* is a sequence of symbols. A *language*, or a *property*, is a set of words. A trace $\sigma \in \Sigma^\infty$ *satisfies* a property $\mathcal{L} \subseteq \Sigma^\infty$ if $\sigma \in \mathcal{L}$ or *violates* it if $\sigma \notin \mathcal{L}$.

In this chapter, we use *Finite Automata (FAs)* to represent both regular and ω-regular languages. We use Non-deterministic Büchi Automata (NBAs) to represent ω-regular languages, which accept infinite strings, and NFAs to represent regular languages, which accept finite strings. Both an NBA and an NFA are written $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, where Q is the set of states, Σ is the alphabet, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. The two types of FAs differ in their accepting conditions.

A *path* (or *run*) through an FA $\mathcal{A}$ from a state $q \in Q$ over a word $\sigma \in \Sigma^\infty$ is a sequence of states $\pi = \langle q_1, q_2, \cdots \rangle$ such that $q_1 = q$ and $q_{i+1} \in \delta(q_i, \sigma_i)$. We write $\mathcal{A}(q, \sigma)$ to denote the set of all runs on $\mathcal{A}$ starting at state q with the word σ. The set of all *reachable states* in an FA $\mathcal{A}$ from a starting state q_0 is $\mathcal{R}each(\mathcal{A}, q_0) = \{q \in Q : \exists \sigma \in \Sigma^\infty.\ \exists \pi \in \mathcal{A}(q_0, \sigma).\ q \in \pi\}$.

A finite run on an NFA $\pi = \langle q_1, q_2, \cdots, q_n \rangle$ is considered *accepting* if $q_n \in F$. For an infinite run ρ on an NBA, we use $\mathit{Inf}(\rho) \subseteq Q$ to denote the set of states that are visited infinitely often, and the run is considered *accepting* when $\mathit{Inf}(\rho) \cap F \neq \varnothing$. $\mathrm{L}(\mathcal{A})$ denotes the language accepted by an FA $\mathcal{A}$. The complement or negation of an FA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is written $\overline{\mathcal{A}}$ where $\mathrm{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathrm{L}(\mathcal{A})$ for NFAs and $\mathrm{L}(\overline{\mathcal{A}}) = \Sigma^\omega \setminus \mathrm{L}(\mathcal{A})$ for NBAs.

An NFA is a Deterministic Finite Automaton (DFA) iff $\forall q \in Q.\ \forall \alpha \in \Sigma.\ |\delta(q, \alpha)| = 1$. Given a DFA $(Q, \Sigma, q_0, \delta, F)$, a state $q \in Q$, and a finite string $\sigma \in \Sigma^*$ where $|\sigma| = n$, the terminal (nth) state of the run over σ beginning in q is given by the function $\delta^* : Q \times \Sigma^* \to Q$.

We use LTL formulae throughout the chapter to illustrate examples of properties because it is a common formalism in the RV area. The syntax of these formulae is defined by the following inductive grammar where p is an atomic proposition, U is the *Until* operator ($\varphi\ U \psi$ means ψ must eventually hold and φ must hold until then), and X is the *Next* operator ($X\varphi$ means φ must hold in the next state, which must exist).

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi\ U \varphi$$

We use the following inductive semantics for the infinite case, where $\sigma \in \Sigma^\omega$. The reader should assume the use of infinite-trace semantics unless otherwise specified where LTL is found in this chapter.

$$\begin{array}{lll}
\sigma \models p & \textit{if} & p \in \sigma(1) \\
\sigma \models \neg\varphi & \textit{if} & \sigma \not\models \varphi \\
\sigma \models \varphi \vee \psi & \textit{if} & \sigma \models \varphi \textit{ or } \sigma \models \psi \\
\sigma \models X\varphi & \textit{if} & \sigma^2 \models \varphi \\
\sigma \models \varphi\ U \psi & \textit{if} & \exists k \geq 1.\ \sigma^k \models \psi \wedge \forall j.\ 1 \leq j < k.\ \sigma^j \models \varphi
\end{array}$$

The language of an LTL formula φ is given in the infinite case by $L[\![\varphi]\!] = \{\sigma \in \Sigma^\omega : \sigma \models \varphi\}$.

For the finite case, where $\sigma \in \Sigma^*$, we use the following inductive semantics.

$$\begin{array}{lll}
\sigma \models p & \textit{if} & |\sigma| > 0 \textit{ and } p \in \sigma(1) \\
\sigma \models \neg\varphi & \textit{if} & \sigma \not\models \varphi \\
\sigma \models \varphi \vee \psi & \textit{if} & \sigma \models \varphi \textit{ or } \sigma \models \psi \\
\sigma \models X\varphi & \textit{if} & |\sigma| > 0 \textit{ and } \sigma^2 \models \varphi \\
\sigma \models \varphi\ U \psi & \textit{if} & \exists k \geq 1.\ \sigma^k \models \psi \wedge \forall j.\ 1 \leq j < k.\ \sigma^j \models \varphi
\end{array}$$

The language of an LTL formula φ is given in the finite case by $L_F[\![\varphi]\!] = \{\sigma \in \Sigma^* : \sigma \models \varphi\}$.

For both infinite and finite-trace semantics we also define the standard notation: $\mathit{true} = p \vee \neg p$ for any proposition p, $\mathit{false} = \neg \mathit{true}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $F\varphi = \mathit{true}\, U\varphi$ (eventually φ), and $G\varphi = \neg F \neg\varphi$ (globally φ).

Example: Consider an infinite trace σ where p holds for the entire trace except the tenth symbol, which is the only symbol where q holds. The LTL formula Gp is violated for σ in the infinite case and it is violated in the finite case for prefixes of σ of at least length ten. The formula is satisfied, however, in the finite case for prefixes of σ of length less than ten. Likewise, the LTL formula Fq is satisfied for σ in the infinite case and in the finite case for prefixes of σ of at least length ten. It is violated for prefixes of σ of length less than ten.

4.3 Foundations of Monitoring

In this section, we establish definitions from previous works referenced in the article. We begin with the truth domains we use and how they relate to monitoring. We then provide traditional definitions of monitorability.

4.3.1 Uncertainty

In RV, there are two prevailing options for checking that a trace of a program's execution satisfies a property: offline and online. In offline RV, we consider a finite trace produced by a program that has terminated. In this case, properties are specified as languages of finite words, for example using a finite-trace semantics to interpret LTL formulae. In online RV, we consider a continuously expanding finite prefix produced by a running program. In this case, properties are specified as languages of infinite words, for example using an infinite-trace semantics to interpret LTL formulae.

In this chapter, we are interested in checking finite prefixes of execution traces against properties specified as languages of infinite words. We say a finite string *determines* inclusion in (or exclusion from) a language of infinite words only if all infinite extensions of the prefix are in (or out of) the language. If some infinite extensions are in the language and some are out, then the finite prefix does not determine inclusion and the result is uncertainty. The problem appears with an LTL property such as Fa, which is satisfied if an a appears in the string. However, if no a has yet been observed, and the program is still executing, it is unknown if the specification will be satisfied in the future.

To express notions of uncertainty in monitoring languages of infinite words, extensions to the Boolean truth domain $\mathbb{B}_2 = \{\top, \bot\}$ have been proposed. $\mathbb{B}_3$ adds a third verdict of *?* to the traditional Boolean notion of *true* or *false* to represent the idea that the specification is neither satisfied nor violated by the current finite prefix [30]. $\mathbb{B}_4$ replaces *?* with *presumably true* ($\top_p$) and *presumably false* ($\bot_p$) to provide more information on what has already been seen [32].

The verdicts $\top_p$ and $\bot_p$ differentiate between prefixes that would satisfy or violate the property interpreted with finite trace semantics. The intuition is that $\bot_p$ indicates that something is required to happen in the future, while $\top_p$ means there is no such outstanding event. For example, if the formula $G(a \to Fb)$ is interpreted as four-value LTL (LTL$_4$) (also called Runtime Verification LTL (RV-LTL) [32], which uses $\mathbb{B}_4$), the verdict on a trace $\langle\{c\}\rangle$ is $\top_p$ because a has not occurred, and therefore no b is required, while the verdict on $\langle\{a\}\rangle$ is $\bot_p$ because there is an a but as yet no b. If the same property is interpreted as Three-value LTL (LTL$_3$) (which uses $\mathbb{B}_3$) the verdicts on both traces would be *?*.

The above intuitions are formalized in Definition 18. Here, we define a property $\mathcal{L}$ to be a set of both finite and infinite traces. The infinite words determine the permanent verdicts of $\top$ and $\bot$ while the finite words are used in the $\mathbb{B}_4$ case to choose between $\top_p$ and $\bot_p$. For both $\mathbb{B}_3$ and $\mathbb{B}_4$, Definition 18 includes a function that evaluates a finite trace prefix with respect to $\mathcal{L}$.

Definition 18 (Evaluation Functions)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$ for each of the truth domains $\mathbb{V} \in \{\mathbb{B}_3, \mathbb{B}_4\}$, we define evaluation functions of the form $\mathfrak{E}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$ as the following.*

For $\mathbb{B}_3 = \{\bot, ?, \top\}$,

$$\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = \begin{cases} \bot & \text{if } \sigma \cdot \mu \notin \mathcal{L}\ \forall \mu \in \Sigma^\omega \\ \top & \text{if } \sigma \cdot \mu \in \mathcal{L}\ \forall \mu \in \Sigma^\omega \\ ? & \text{otherwise} \end{cases}$$

For $\mathbb{B}_4 = \{\bot, \bot_p, \top_p, \top\}$,

$$\mathfrak{E}_{\mathbb{B}_4}(\mathcal{L})(\sigma) = \begin{cases} \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) & \text{if } \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) \neq ? \\ \bot_p & \text{if } \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = ? \text{ and } \sigma \notin \mathcal{L} \\ \top_p & \text{if } \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = ? \text{ and } \sigma \in \mathcal{L} \end{cases}$$

Example: Suppose we would like to monitor the LTL formula $\varphi = G(a) \vee b$ using the $\mathbb{B}_4$ truth domain. The language (property) to monitor is $\mathcal{L} = L[\![\varphi]\!] \cup L_F[\![\varphi]\!]$. The following are the evaluations for given finite prefixes:

$\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{b\}\rangle)$	$= \top$	All infinite strings beginning with this prefix are in the language.
$\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{\}\rangle)$	$= \bot$	No infinite strings beginning with this prefix are in the language.
$\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\}\rangle)$	$= \top_p$	Some infinite strings beginning with this prefix are in the language, and the finite prefix is itself in the language (because $\langle\{a\}\rangle \in L_F[\![\varphi]\!]$).

Monitors also exist for properties that cannot be specified in LTL or other common temporal logics. This chapter uses language-theoretic formalisms that allow for the monitoring of any language of finite and infinite words. For example, it is possible to monitor a property consisting of an infinite repetition of every valid C program. Clearly, such a language is not representable in LTL since recognizing it requires a stack.

For the verdicts specified in Definition 18 for $\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}$ to make intuitive sense, the infinite and finite words in the language must be related. For an LTL formula φ, the infinite words are defined by $L[\![\varphi]\!]$ and the finite words by $L_F[\![\varphi]\!]$. Given a language of finite words, Falcone et al. defined how to construct both the finite-words and infinite-words in [85]. In the general case, however, the precise relationship between the two subsets has not been defined. This relationship remains a subject for future work on monitoring non-ω-star-free languages.

Introducing the idea of uncertainty in monitoring causes the possibility that some properties might never reach a definite, *true* or *false* verdict. A monitor that will only ever return a *?* result does not have much utility. The *monitorability* of a property captures this notion of the reachability of definite verdicts.

4.3.2 Monitorability

In this section, we examine the four most common definitions of monitorability. To define monitorability for properties over unreliable channels, we must first define monitorability for properties over ideal channels. Rather than choose one definition, we introduce established definitions and allow the reader to select that of their preference.

We begin with the definition of σ-Monitorability, which depends not only on the monitored property but also on the already-seen trace prefix. For each definition of monitorability that depends only on the monitored property $\mathbb{M} \in \{$C(lassical), W(eak), A(lternative)$\}$, we introduce an evaluation predicate of the form $\boldsymbol{M}_{on}^{\mathbb{M}} : 2^{\Sigma^\infty} \to \mathbb{B}_2$ that returns *true* iff the input property is monitorable. We say that an LTL formula φ is monitorable if its language $L[\![\varphi]\!] \cup L_F[\![\varphi]\!]$ is monitorable.

σ-Monitorability

Pnueli and Zaks introduced the first formal definition of monitorability in their work on Property Specification Language (PSL) for model checking in 2006 [185]. They define monitorable properties given a trace prefix σ. Subsequent works all define monitorability for a property without assuming knowledge of any part of the trace.

Definition 19 (σ-Monitorability)**.** *Given a finite sequence $\sigma \in \Sigma^*$, a property $\mathcal{L} \subseteq \Sigma^\infty$ is σ-monitorable iff $\exists \eta \in \Sigma^* .\ \forall s \in \Sigma^\omega .\ (\sigma \cdot \eta \cdot s \models \mathcal{L}$ or $\sigma \cdot \eta \cdot s \not\models \mathcal{L})$.*

That is, there exists another finite sequence η such that $\sigma \cdot \eta$ determines inclusion in or exclusion from $\mathcal{L}$.

For example, the LTL formula GFp is non-σ-monitorable for any finite prefix, because the trace needed to determine the verdict must be infinite. Other properties are σ-monitorable for some prefixes but not others. For example, there is no point to continuing to monitor $GFp \vee q$ if q does not hold in the first symbol of the trace.

Classical Monitorability

Bauer, Leuker, and Schallhart reformulated this definition of monitorability and proved that safety (e.g. Gp) and guarantee (e.g. Fp) properties represent a proper subset of the class of monitorable properties [33]. It was already known that the class of monitorable properties was not limited to safety and guarantee properties from the work of d'Amorim and Roşu on monitoring ω-regular languages [58], however that work did not formally define monitorability. Diekert and Leuker have also defined a purely topological version of this definition of monitorability [67].

The definition of monitorability given by Bauer et al. is identical to Definition 19 except that it considers all possible trace prefixes instead of a specific prefix [84, 85] and it excludes languages with finite words. The restriction to infinite words is due to their interest in defining monitorable LTL$_3$ properties, which only considers infinite traces.

Bauer et al. use Kupferman and Vardi's definitions of *good* and *bad* prefixes of an infinite trace [140] to define what they call an *ugly* prefix. That is, given a language of infinite strings $\mathcal{L} \subseteq \Sigma^\omega$,

- a finite word $b \in \Sigma^*$ is a *bad prefix* for $\mathcal{L}$ iff $\forall s \in \Sigma^\omega .\ b \cdot s \notin \mathcal{L}$, and
- a finite word $g \in \Sigma^*$ is a *good prefix* for $\mathcal{L}$ iff $\forall s \in \Sigma^\omega .\ g \cdot s \in \mathcal{L}$.

Bauer et al. use good and bad prefixes to define *ugly* prefixes and then use ugly prefixes to define Classical Monitorability.

Definition 20 (Ugly Prefix)**.** *Given a language of infinite strings $\mathcal{L} \subseteq \Sigma^\omega$, a finite word $u \in \Sigma^*$ is an* ugly prefix *for $\mathcal{L}$ iff $\nexists s \in \Sigma^*.\ u \cdot s$ is either a good or bad prefix.*

Definition 21 (Classical Monitorability)**.** *Given a language of infinite strings $\mathcal{L} \subseteq \Sigma^\omega$,*

$$\boldsymbol{M}_{on}^{C}(\mathcal{L}) = \nexists u \in \Sigma^*.\ u \textit{ is an ugly prefix for } \mathcal{L}$$

Many works have explored decision procedures for Classical Monitorability. Diekert, Muscholl, and Walukiewicz proved that the problem is PSPACE-Hard and can be solved in EXPSPACE [68] for ω-regular languages. This result was most recently refined by Peled and Havelund, who showed that deciding Classical Monitorability for these languages is EXPSPACE-Complete [178].

Weak Monitorability

Recently, both Chen et al. [49] and Peled and Havelund [178] proposed a weaker definition of monitorability that includes more properties than the Classical definition. They observed that there are properties that are classically non-monitorable, but that are still useful to monitor. For example, $\neg \boldsymbol{M}_{on}^{C}(L[\![a \wedge GFa]\!])$ because any trace that begins with a must then satisfy or violate GFa, which is not possible. However, $a \wedge GFa$ is violated by traces that do not begin with a, so it may have some utility to monitor.

Definition 22 (Weak Monitorability)**.** *Given a language of infinite strings $\mathcal{L} \subseteq \Sigma^\omega$,*

$$\boldsymbol{M}_{on}^{W}(\mathcal{L}) = \exists p \in \Sigma^*.\ p \textit{ is not an ugly prefix for } \mathcal{L}$$

Deciding that an ω-regular property is Weakly Monitorable requires testing that no information may be obtained from the monitor. Peled and Havelund gave an algorithm for deciding Weak Monitorability for these languages and showed that it is EXPSPACE-Complete [178].

Alternative Monitorability

Falcone et al. observed that the class of monitorable properties should depend on the truth domain of the monitored formula. However, they noticed that changing from $\mathbb{B}_3$ to $\mathbb{B}_4$ does not influence the set of monitorable properties under classical monitorability [84, 85].

To resolve this perceived shortcoming, the authors of [84, 85] introduce an *alternative* definition of monitorability. They introduce the notion of an *r-property* (runtime property) which separates the property's language of finite and infinite traces into disjoint sets. We do not require this distinction and treat the property as a single set containing both finite and infinite traces. Falcone et al. then define an alternative notion of monitorability for a property using a variant of Definition 18.

Definition 23 (Alternative Monitorability)**.** *Given a truth domain* $\mathbb{V}$ *and an evaluation function for* $\mathbb{V}$*,* $\boldsymbol{\mathfrak{E}}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$ *and a property* $\mathcal{L} \subseteq \Sigma^\infty$*,*

$$\boldsymbol{Mon}^{A}(\mathcal{L}) = \forall \sigma_{in} \in \mathcal{L} \cap \Sigma^* \boldsymbol{.}\ \forall \sigma_{out} \notin \mathcal{L} \cap \Sigma^* \boldsymbol{.}\ \boldsymbol{\mathfrak{E}}_{\mathbb{V}}(\mathcal{L})(\sigma_{in}) \neq \boldsymbol{\mathfrak{E}}_{\mathbb{V}}(\mathcal{L})(\sigma_{out})$$

Definition 23 says that, given a truth domain, a property with both finite and infinite words is monitorable if evaluating the finite strings *in* the property always yield different verdicts from evaluating the finite strings *out* of the property. By Definition 23, only properties with finite words are considered monitorable and its results must be understood in the same context as $\boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}$, where finite words identify prefixes where no outstanding event precludes satisfaction.

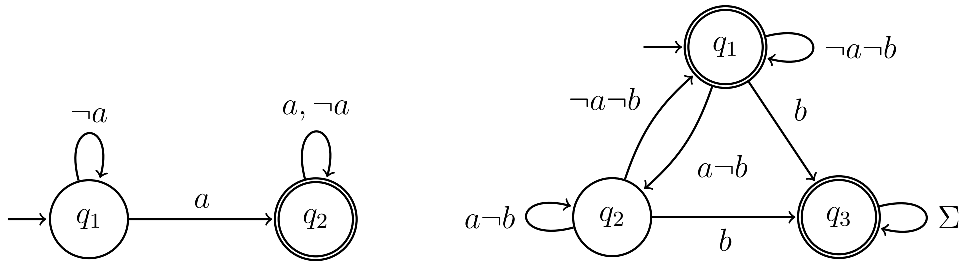
Procedures for deciding if an ω-regular property is Alternatively Monitorable depend on the truth domain. For $\mathbb{B}_3$, monitorable properties are exactly the union of *Safety* and *Guarantee* properties (see Section 4.8) [85]. Determining inclusion in these classes is known to be PSPACE-Complete [207]. For $\mathbb{B}_4$, monitorable properties are the *Reactivity* properties, which are all properties representable in LTL [85]. Deciding if a language represented as an NBA is a Reactivity property is PSPACE-Complete [66].

4.4 Unreliable Channels

For a property to be monitorable over an unreliable channel it must be monitorable over ideal channels, and it must reach the correct verdict despite the unreliable channel. To illustrate this, we introduce an example.

4.4.1 An Example with Unreliable Channels

Consider the LTL formula $\varphi = Fa$ over the alphabet $\Sigma = \{\{a\}, \{\neg a\}\}$. That is, all traces that contain at least one symbol with a satisfy φ. We assume that the trace is monitored remotely, and, for this example, we will adopt a $\mathbb{B}_3$ truth domain. Using $\boldsymbol{\mathfrak{E}}_{\mathbb{B}_3}$ from Definition 18, the verdict on finite prefixes without an a, is *?*, while the verdict when an a is included is $\top$. Figure 4.1a shows the NBA for such a property.



(a) NBA for Fa (b) NBA for $G(a \to F\neg a) \vee Fb$

Figure 4.1: Büchi automata for example properties

Monitorability under reordering

Suppose that the channel over which the trace is transmitted may reorder events. That is, events are guaranteed to be delivered, but not necessarily in the same order in which they were sent.

We argue that Fa should be considered monitorable over a channel that reorders the trace. First, the property is monitorable over an ideal channel (see Section 4.3.2). Second, given any trace prefix, reordering the prefix would not change the verdict of a monitor. Any a in the trace will cause a transition to state q_2, regardless of its position.

Note that we are not concerned with *when* the verdict occurs. For example, assume a trace $\langle\{a\}, \{\neg a\}\rangle$ that is reordered to $\langle\{\neg a\}, \{a\}\rangle$. Both traces result in a $\mathbb{B}_3$ verdict of $\top$, but in the reordered case it comes one symbol later. This article considers these results to be equivalent, but future work could consider the implications of such a change in timing.

Monitorability under loss

Now suppose that, instead of reordering, the channel over which the trace is transmitted may lose events. That is, the order of events is guaranteed to be maintained, but some events may be missing from the trace observed by the monitor.

We argue that Fa should not be considered monitorable over a channel that loses events, even though the property is deemed to be monitorable over an ideal channel. It is possible for the verdict from the monitor to be different from what it would be given the original trace. For example, assume a trace $\langle\{a\}, \{\neg a\}\rangle$. For this trace, the verdict from an LTL$_3$ monitor would be $\top$. However, if the first symbol (containing a) is lost, the verdict would be *?*.

Note that there may still be some utility to monitor Fa when symbols may be lost because a $\top$ verdict is actionable. That is, if the monitor receives a trace $\langle\{a\}\rangle$ then a must have held in the original trace as well. In this case, we call $\top$ a *trustworthy* verdict. We explore the concept of trustworthy verdicts in Section 4.6.

4.4.2 Trace Mutations

To model unreliable channels, we introduce *trace mutations*. A mutation represents the possible modifications to traces from communication over unreliable channels. These mutations are defined as relations between unmodified original traces and their mutated counterparts. Trace mutations include only finite traces because only finite prefixes may be mutated in practice.

There are four trace mutations $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$ where $\mathcal{M}$ denotes any of the relations in Definitions 24, 25, 26, and 27 or a union of any number of them, and k denotes the number of inductive steps.

Definition 24 (Loss Mutation). $\mathit{Loss} = \{(\sigma, \sigma') : \sigma = \sigma' \vee$
$\exists \alpha, \beta \in \Sigma^* \mathbf{.}\ \exists x \in \Sigma \mathbf{.}$
$\sigma = \alpha \cdot \langle x \rangle \cdot \beta \wedge \sigma' = \alpha \cdot \beta\}$

Definition 25 (Corruption Mutation). $\mathit{Corruption} = \{(\sigma, \sigma') :$
$\exists \alpha, \beta \in \Sigma^* \mathbf{.}\ \exists x, y \in \Sigma \mathbf{.}$
$\sigma = \alpha \cdot \langle x \rangle \cdot \beta \wedge \sigma' = \alpha \cdot \langle y \rangle \cdot \beta\}$

Definition 26 (Stutter Mutation). $\mathit{Stutter} = \{(\sigma, \sigma') : \sigma = \sigma' \vee$
$\exists \alpha, \beta \in \Sigma^* \mathbf{.}\ \exists x \in \Sigma \mathbf{.}$
$\sigma = \alpha \cdot \langle x \rangle \cdot \beta \wedge \sigma' = \alpha \cdot \langle x, x \rangle \cdot \beta\}$

Definition 27 (Out-of-Order Mutation). $\mathit{OutOfOrder} = \{(\sigma, \sigma') :$
$\exists \alpha, \beta \in \Sigma^* \mathbf{.}\ \exists x, y \in \Sigma \mathbf{.}$
$\sigma = \alpha \cdot \langle x, y \rangle \cdot \beta \wedge \sigma' = \alpha \cdot \langle y, x \rangle \cdot \beta\}$

Definition 28 (Inductive k-Mutations). *Given any mutation or union of mutations $\mathcal{M}$, we define $\mathcal{M}^k$ inductively as the following.*

$$\mathcal{M}^1 \in \{\ \bigcup m : m \in 2^{\{\mathit{Loss}, \mathit{Corruption}, \mathit{Stutter}, \mathit{OutOfOrder}\}} \wedge m \neq \varnothing\ \ \}$$

$$\mathcal{M}^{k+1} = \mathcal{M}^k \cup \{\ \ (\sigma_1, \sigma_3) : \exists \sigma_2 \mathbf{.}\ (\sigma_1, \sigma_2) \in \mathcal{M}^k \wedge (\sigma_2, \sigma_3) \in \mathcal{M}^1\ \ \}$$

These mutations are based on Lozes and Villard's interference model [156]. Other works on the verification of unreliable channels, such as [43], have chosen to include *insertion* errors instead of *Corruption* and *OutOfOrder*. We prefer to define *Corruption* and *OutOfOrder* because the mutations more closely reflect our real-world experiences. For example, packets sent using the UDP may be corrupted or arrive out-of-order, but packets must be sent before these mutations occur.

In this thesis, we assume that the monitor has no information about how a received trace has been modified by an unreliable channel. Instead, we only permit that the channel is known to sometimes mutate traces in a certain manner (e.g. losing symbols). This differs from and is a weaker assumption than some other works, where trace modifications are marked [95, 27, 122, 148].

We say a mutation M is *prefix-assured* when $\forall(\sigma, \sigma') \in M$ such that $|\sigma| > 1$, $\exists(\sigma_p, \sigma'_p) \in M$, where $\sigma_p \sqsubseteq \sigma$ and $\sigma'_p \sqsubseteq \sigma'$. All mutations $\mathcal{M}^1$ are prefix-assured. Combining mutations is possible under Definition 28, and it is possible to form any combination of strings by doing so. This capability is important to ensure the mutation model is complete.

Definitions 24 through 27 include every possible mutation. That is, it is possible to apply a combination of these mutations to a trace to transform it into any other trace.

Theorem 1 (Completeness of Mutations). *Given any two sets of non-empty traces* $S, S' \subseteq \Sigma^* \setminus \{\varepsilon\}$, $\exists k \in \mathbb{N}$. $(\mathit{Loss} \cup \mathit{Corruption} \cup \mathit{Stutter})^k = S \times S'$.

Proof: First, Definition 25 allows an arbitrary symbol in a string to be changed to any other symbol. Thus, $\forall\sigma' \in \Sigma^*$ there exists $\sigma : (\sigma, \sigma') \in \mathit{Corruption}^n$ where $|\sigma| = |\sigma'|$ and $n \geq |\sigma|$. A string can also be lengthened or shortened arbitrarily, so long as it is non-empty. Definition 26 allows lengthening, because $\mathit{Stutter}(\sigma, \sigma') \implies |\sigma| < |\sigma'|$, while Definition 24 allows shortening, because $\mathit{Loss}(\sigma, \sigma') \implies |\sigma| > |\sigma'|$. □

These mutations are general and it may be useful for practitioners to define their own, more constrained mutations based on domain knowledge. For example, if a communications protocol guarantees delivery of high priority messages but allows low priority messages to be lost, this can be modeled as a mutation. Some properties may be monitorable over this more-precise mutation when they would not be monitorable over the *Loss* mutation, which permits losing any message.

Even Definition 27 (*OutOfOrder*) is a more-constrained version of the *Corruption* mutation. That is, $\mathit{OutOfOrder}^n \subset \mathit{Corruption}^{2n}\ \forall n \in \mathbb{N}$. *OutOfOrder* is unnecessary for the completeness of the mutation model, as can be seen in Theorem 1. However, we consider the mutation to be general enough to include here, and a combination of Definitions 24, 25, and 26 can only over-approximate the *OutOfOrder* relation.

4.5 Immunity to Trace Mutations

The two requirements for a property to be monitorable over an unreliable channel are that the property is monitorable over an ideal channel and that the property is *immune* to the effects of the unreliable channel. A monitor must be able to reach a meaningful, actionable verdict for a trace prefix, and the verdict must also be *correct*. If a monitored property is immune to a mutation then we can trust the monitor's verdict whether or not the observed trace is mutated.

The notion of immunity to a mutation is related to the concept of monotonicity of entailment of a logical system. For a monotonic logic, anything that could be concluded before information is added can still be concluded after. In this case, however, mutations to a trace may remove or modify information as well as add. Monotonicity of a property with regard to past events was also previously defined by Joshi, Tchamgoue, and Fischmeister for channels with *Loss* to mean that the property's monitor cannot change its verdict if lost information is added to the trace [122]. Monotonicity has also been used in RV in the sense of a monotone function to describe how verdicts like $\top$ and $\bot$ may not change once reached [50]. Here, we use the term immunity to avoid overloading the word monotonic further in the field of RV.

Definition 29 characterizes properties where the given trace mutation will have no effect on the evaluation verdict. For example, the LTL formula Fa from Figure 4.1a is immune to $\mathit{OutOfOrder}^1$ (an LTL formula φ is immune to a mutation $\mathcal{M}^k$ if its language $L[\![\varphi]\!] \cup L_F[\![\varphi]\!]$ is immune to $\mathcal{M}^k$) with truth domain $\mathbb{B}_3$ or $\mathbb{B}_4$ because reordering the input trace cannot change the verdict.

Definition 29 (Full Immunity to Unreliable Channels)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, a truth domain $\mathbb{V}$, and an evaluation function $\boldsymbol{\mathfrak{E}}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$, $\mathcal{L}$ is immune to $\mathcal{M}^k$ iff $\forall (\sigma, \sigma') \in \mathcal{M}^k \,\boldsymbol{.}\, \boldsymbol{\mathfrak{E}}_{\mathbb{V}}(\mathcal{L})(\sigma) = \boldsymbol{\mathfrak{E}}_{\mathbb{V}}(\mathcal{L})(\sigma')$.*

Example: We want to check if the LTL formula $\varphi = Ga$ is immune to the $\mathit{Stutter}^1$ mutation for truth domain $\mathbb{B}_4$. The property for this formula is $\mathcal{L} = L[\![\varphi]\!] \cup L_F[\![\varphi]\!]$. $\mathcal{L}$ is immune to $\mathit{Stutter}^1$ for $\mathbb{B}_4$ iff the verdict from $\boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}$ is always the same when applied to both the left and right sides of every pair in $\mathit{Stutter}^1$. Where $\Sigma = \{\{a\}, \{\neg a\}\}$, we check the following:

- $(\boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\}\rangle), \boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\}\rangle)) = (\top_p, \top_p)$
- $(\boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{\neg a\}\rangle), \boldsymbol{\mathfrak{E}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{\neg a\}\rangle)) = (\bot, \bot)$

- $(\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\}\rangle), \boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\},\{a\}\rangle)) = (\top_p, \top_p)$
- $(\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{\neg a\}\rangle), \boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{\neg a\},\{\neg a\}\rangle)) = (\bot, \bot)$
- $(\boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\},\{a\}\rangle), \boldsymbol{\mathfrak{L}}_{\mathbb{B}_4}(\mathcal{L})(\langle\{a\},\{a\},\{a\}\rangle)) = (\top_p, \top_p)$
- ...

If every pair has an equal verdict, then $\mathcal{L}$ (and φ) is immune to $\mathit{Stutter}^1$ for $\mathbb{B}_4$.

Definition 29 specifies a k-Mutation from Definition 28, but a property that is immune to a mutation for some k is immune to that mutation for *any* k. This significant result forms the basis for checking for mutation immunity in Section 4.10. The intuition is that, since we assume any combination of symbols in the alphabet is a possible ideal trace, and a mutation could occur at any time, one mutation is enough to violate immunity for any vulnerable property.

Theorem 2 (Single Mutation Immunity Equivalence)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, a trace mutation $\mathcal{M} \subseteq \Sigma^* \times \Sigma^*$, and a number of applications of that mutation k, $\mathcal{L}$ is immune to $\mathcal{M}^k$ iff $\mathcal{L}$ is immune to $\mathcal{M}^1$.*

Proof: Since k-Mutations are defined inductively, Theorem 2 is equivalent to the statement that $\mathcal{L}$ is immune to $\mathcal{M}^{k+1}$ iff $\mathcal{L}$ is immune to $\mathcal{M}^k$. Now assume by way of contradiction a property $\mathcal{L}_{\text{bad}} \subseteq \Sigma^\infty$ such that $\mathcal{L}_{\text{bad}}$ is immune to some k-Mutation M^k but not to M^{k+1}. That is, given a truth domain $\mathbb{V}$, there exists a pair of traces $(\sigma_1, \sigma_3) \in M^{k+1}$ such that $\boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_1) \neq \boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_3)$.

From Definition 28, either $(\sigma_1, \sigma_3) \in M^k$, or there exists both $(\sigma_1, \sigma_2) \in \mathcal{M}^k$ and $(\sigma_2, \sigma_3) \in \mathcal{M}^1$ such that $\boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_1) \neq \boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_3)$. It cannot be true that $(\sigma_1, \sigma_3) \in M^k$ since $\mathcal{L}_{\text{bad}}$ is immune to M^k so there must exist pairs $(\sigma_1, \sigma_2) \in \mathcal{M}^k$ and $(\sigma_2, \sigma_3) \in \mathcal{M}^1$. Since $\mathcal{L}_{\text{bad}}$ is immune to $\mathcal{M}^k$, $\boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_1) = \boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_2)$ so it must be true that $\boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_2) \neq \boldsymbol{\mathfrak{L}}_{\mathbb{V}}(\mathcal{L}_{\text{bad}})(\sigma_3)$. However, it is clear from Definition 28 that $M^k \subseteq M^{k+1}$, so $M^1 \subseteq M^k$ for any k, which is a contradiction.

For the reverse case, assume a property $\mathcal{L}_{\text{sad}} \subseteq \Sigma^\infty$ such that $\mathcal{L}_{\text{sad}}$ is not immune to some k-Mutation M^k but is immune to M^{k+1}. However, as we saw before, $M^k \subseteq M^{k+1}$ so $\mathcal{L}_{\text{sad}}$ must not be immune to M^{k+1}, a contradiction. □

Immunity under Definition 29 is too strong to be a requirement for monitorability over an unreliable channel, however. Take, for example, the property $G(a \to F\neg a) \vee Fb$, as shown in Figure 4.1b. By Definition 29 with truth domain $\mathbb{B}_4$ this property is vulnerable (not immune) to *OutOfOrder*[1] because reordering symbols may change the verdict. For example, the trace $\langle\{a, \neg b\}, \{\neg a, \neg b\}\rangle$ results in a verdict of $\top_p$, but reordering the trace to $\langle\{\neg a, \neg b\}, \{a, \neg b\}\rangle$ changes the verdict to $\bot_p$. However, this property is monitorable under all definitions in Section 4.3.2, because it is always possible to reach a $\top$ verdict if a b appears. We would like a modified definition of immunity that only considers the parts of a property that affect its monitorability.

To achieve this modified definition of immunity, we consider only the determinization of the property to be crucial. Definition 30 characterizes properties for which satisfaction and violation are unaffected by a mutation. We call this *true-false immunity*, and it is equivalent to immunity with truth domain $\mathbb{B}_3$. The intuition is that $\mathbb{B}_3$ treats all verdicts outside $\{\top, \bot\}$ as the symbol *?* so immunity with this truth domain does not concern non-*true-false* verdicts.

Definition 30 (True-False Immunity to Unreliable Channels)**.** *Given a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, a language $\mathcal{L} \subseteq \Sigma^\infty$ is true-false immune to $\mathcal{M}^k$ iff $\mathcal{L}$ is immune to $\mathcal{M}^k$ for the truth domain $\mathbb{B}_3$.*

The true-false immunity of a property to a mutation is necessary but not sufficient to show that the property is monitorable over an unreliable channel. For example, the LTL formula GFa is true-false immune to all mutations because $\mathfrak{E}_{\mathbb{B}_3}(L[\![GFa]\!])(\sigma) = \textit{?}$ for any prefix $\sigma \in \Sigma^*$, but the property is not monitorable. We can now define monitorability over unreliable channels in the general case.

Definition 31 (Monitorability over Unreliable Channels)**.** *Given a language $\mathcal{L} \subseteq \Sigma^\infty$, a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, and a definition of monitorability $\boldsymbol{Mon}^{\mathbb{M}} : 2^{\Sigma^\infty} \to \mathbb{B}_2$, $\mathcal{L}$ is monitorable over $\mathcal{M}^k$ iff $\boldsymbol{Mon}^{\mathbb{M}}(\mathcal{L})$ and $\mathcal{L}$ is true-false immune to $\mathcal{M}^k$.*

The question of what languages are considered monitorable by Definitions 21, 22, and 23 has largely been answered by prior work. To understand what languages are monitorable over an unreliable channel, we must understand what languages are true-false immune to the given mutation.

4.6 Trustworthy Verdicts

Some properties that are unmonitorable over an unreliable channel may still have some utility. A property that is not true-false immune to a trace mutation may still yield *trustworthy* verdicts when monitored. This idea is similar to that of weak-monitorability, defined in Section 4.3.2, in that some properties may be interesting to monitor despite being classically unmonitorable. In this section we define trustworthy verdicts and examine their practical consequences.

A trustworthy verdict for a property over an unreliable channel implies the same verdict for the property over an ideal channel. For example, $\mathfrak{E}_{\mathbb{B}_3}(L[\![Fa]\!])(\sigma) = \top$ (the NBA for the LTL formula Fa is shown in Figure 4.1a) when there exists a symbol in σ where a holds. Over a channel with the *Loss* mutation, a $\top$ verdict guarantees that a held in the original as well as the mutated trace, since *Loss* cannot *add* such a symbol.

Definition 32 (Trustworthy Verdicts)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, a truth domain $\mathbb{V}$, and an evaluation function $\mathfrak{E}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$, a verdict $v \in \mathbb{V}$ is trustworthy for $\mathcal{L}$ over a channel with $\mathcal{M}^k$ iff $\forall(\sigma, \sigma') \in \mathcal{M}^k$. $(\mathfrak{E}_{\mathbb{V}}(\mathcal{L})(\sigma') = v) \to (\mathfrak{E}_{\mathbb{V}}(\mathcal{L})(\sigma) = v)$.*

Definition 32 specifies a k-Mutation from Definition 28, but a property that is immune to a mutation for some k is immune to that mutation for *any* k. This result follows from Theorem 2, which specifies single mutation immunity equivalence.

Corollary 1 (Single Mutation Trustworthy Verdict Equivalence)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, a truth domain $\mathbb{V}$, a trace mutation $\mathcal{M} \subseteq \Sigma^* \times \Sigma^*$, and a number of applications of that mutation k, a verdict $v \in \mathbb{V}$ is trustworthy from $\mathcal{L}$ over a channel with $\mathcal{M}^k$ iff v is trustworthy for $\mathcal{L}$ over a channel with $\mathcal{M}^1$.*

Proof: Corrollary 1 is implied by Theorem 2. Theorem 2 specifies that a property $\mathcal{L} \subseteq \Sigma^\infty$ is immune to a mutation $M^k \subseteq \Sigma^* \times \Sigma^*$ iff $\mathcal{L}$ is immune to M^1. By Definition 29, if the property is immune to M^1 for a truth domain $\mathbb{V}$ and an evaluation function $\mathfrak{E}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$ then for all pairs $(\sigma, \sigma') \in \mathcal{M}^k$ and for all verdicts $v \in \mathbb{V}$ $(\mathfrak{E}_{\mathbb{V}}(\mathcal{L})(\sigma) = v) \leftrightarrow (\mathfrak{E}_{\mathbb{V}}(\mathcal{L})(\sigma') = v)$. Therefore the same result applies for a specific verdict $v \in \mathbb{V}$ and one-way implication instead of two. □

If all verdicts in a truth domain are trustworthy for a property and a trace mutation, then that property is immune to the trace mutation. This equivalence allows us to apply the study of trustworthy verdicts to that of mutation immunity. In Section 4.8, we classify properties with trustworthy verdicts over unreliable channels which applies equally to the classification of mutation-immune properties.

Theorem 3 (Trustworthy Verdict Immunity Equivalence)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, a truth domain $\mathbb{V}$, and an evaluation function $\mathfrak{E}_{\mathbb{V}} : 2^{\Sigma^\infty} \to \Sigma^* \to \mathbb{V}$, $\mathcal{L}$ is immune to $\mathcal{M}^k$ iff all verdicts in $\mathbb{V}$ are trustworthy over a channel with $\mathcal{M}^k$.*

Proof: The proof is trivially derived from Definitions 29 and 32. If for all pairs $(\sigma, \sigma') \in \mathcal{M}^k$ and for all verdicts $v \in \mathbb{V}$ it is true that $\mathfrak{L}_{\mathbb{V}}(\mathcal{L})(\sigma') = v$ implies $\mathfrak{L}_{\mathbb{V}}(\mathcal{L})(\sigma) = v$, then for all pairs $(\sigma, \sigma') \in \mathcal{M}^k$ and all verdicts $v \in \mathbb{V}$ it must be true that $\mathfrak{L}_{\mathbb{V}}(\mathcal{L})(\sigma) = \mathfrak{L}_{\mathbb{V}}(\mathcal{L})(\sigma')$. □

Corollary 2 (Trustworthy Verdict True-False Immunity Equivalence)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\infty$, and a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, $\mathcal{L}$ is true-false immune to $\mathcal{M}^k$ iff all verdicts in $\mathbb{B}_3$ are trustworthy over a channel with $\mathcal{M}^k$.*

Proof: The proof follows directly from Definition 30 and Theorem 3. For a property to be true-false immune to a mutation it must be immune for the $\mathbb{B}_3$ truth domain. If all verdicts in a domain are trustworthy for a property and mutation, then the property is immune to that mutation. □

4.7 Classification for Mutation Immunity

In this section, we update the monitorability-focused refinement of the safety-liveness taxonomy, recently introduced by Peled and Havelund [178]. This classification is designed so that its delineations between classes align well with questions of monitorability. This makes it better suited for our purposes than the more established Safety-Progress Hierarchy [45]. We are interested in classifying ω-regular properties that are immune to trace mutations from unreliable channels.

4.7.1 The FR/FS Classification

Peled and Havelund classify properties by whether they are Finitely Refutable (FR) or Finitely Satisfiable (FS) [178]. An ω-regular property $\mathcal{L} \subseteq \Sigma^\omega$ must be one of the following.

- Always Finitely Refutable (`AFR`) iff $\forall \sigma \notin \mathcal{L}.\ \exists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \notin \mathcal{L}$
- Sometimes Finitely Refutable (`SFR`) iff $\exists \sigma \notin \mathcal{L}.\ \exists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \notin \mathcal{L}$
- Never Finitely Refutable (`NFR`) iff $\forall \sigma \notin \mathcal{L}.\ \nexists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \notin \mathcal{L}$

Additionally, $\mathcal{L}$ must be one of the following.

- Always Finitely Satisfiable (AFS) iff $\forall \sigma \in \mathcal{L}.\ \exists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \in \mathcal{L}$
- Sometimes Finitely Satisfiable (SFS) iff $\exists \sigma \in \mathcal{L}.\ \exists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \in \mathcal{L}$
- Never Finitely Satisfiable (NFS) iff $\forall \sigma \in \mathcal{L}.\ \nexists \alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \mu \in \mathcal{L}$

The definitions for AFR, NFR, AFS, and NFS map directly to the classic definitions of safety and liveness properties, and their duals, guarantee and morbidity. The authors of [178] show that all ω-regular properties are included in both $\texttt{AFR} \cup \texttt{SFR} \cup \texttt{NFR}$ and $\texttt{AFS} \cup \texttt{SFS} \cup \texttt{NFS}$.

- **Liveness** (NFR) – A property $\mathcal{L} \subseteq \Sigma^\omega$ is a liveness property iff for all finite prefixes $\alpha \in \Sigma^*$ there exists an infinite suffix $\beta \in \Sigma^\omega$ such that $\alpha \cdot \beta \in \mathcal{L}$.

- **Morbidity** (NFS) – A property $\mathcal{L} \subseteq \Sigma^\omega$ is a morbidity property iff for all finite prefixes $\alpha \in \Sigma^*$ there exists an infinite suffix $\beta \in \Sigma^\omega$ such that $\alpha \cdot \beta \notin \mathcal{L}$.

- **Safety** (AFR) – A property $\mathcal{L} \subseteq \Sigma^\omega$ is a safety property iff for all infinite traces $\sigma \notin \mathcal{L}$ there exists a finite trace $\alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and for all infinite suffixes $\beta \in \Sigma^\omega$ $\alpha \cdot \beta \notin \mathcal{L}$

- **Guarantee** (AFS) – A property $\mathcal{L} \subseteq \Sigma^\omega$ is a guarantee property iff for all traces $\sigma \in \mathcal{L}$ there exists a finite prefix $\alpha \in \Sigma^*$ such that $\alpha \sqsubseteq \sigma$ and for all infinite suffixes $\beta \in \Sigma^\omega$ $\alpha \cdot \beta \in \mathcal{L}$

The FR/FS classification is defined by the intersections beetween pairs of FR and FS classes. These intersections are shown in Figure 4.2, which also labels the intersection $\texttt{SFR} \cap \texttt{SFS}$ as **Quaestio**, which are the ω-regular properties not covered by the liveness, morbidity, safety, and guarantee classes. In the figure, each of NFR, NFS, AFR, and AFS is shown as a stadium shape with their intersections in the corners. The $\texttt{SFR} \cap \texttt{SFS}$ class surrounds the stadia and is also represented in the center of the diagram.

4.7.2 Additional Property Classes

We introduce five classes of properties that overlap with the classes from the FR/FS taxonomy. These are Proximate, Tolerant, Permissive, Inclusive, and Exclusive. We propose language-theoretic definitions for these classes and locate them within the context of the FR/FS framework.

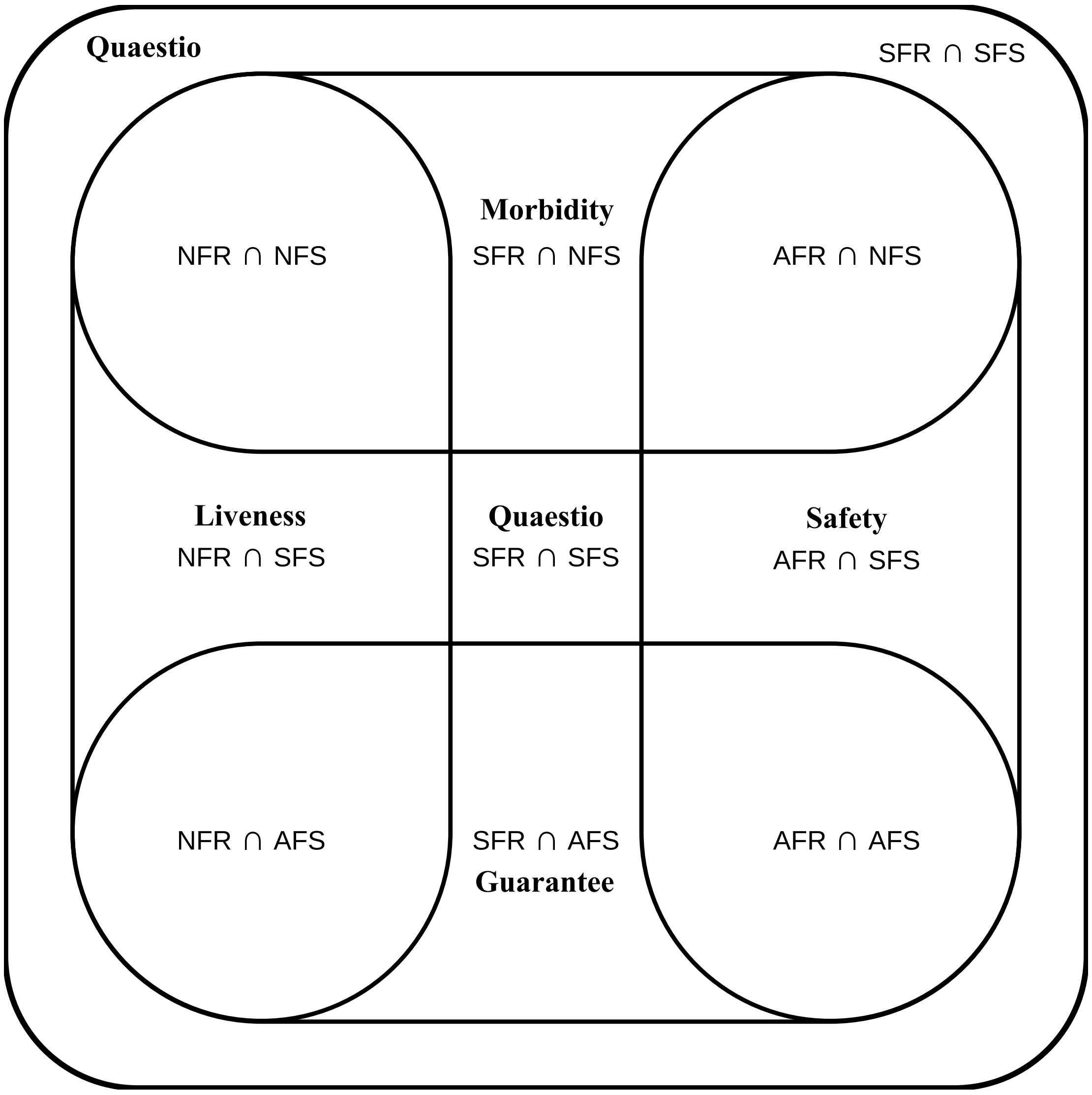


Figure 4.2: The original FR/RS property classification. In the figure, Liveness is `NFR`, Morbidity is `NFS`, Safety is `AFR`, and Guarantee is `AFS`

The FR/FS classification provides the basis for a framework for relating properties that are immune to a trace mutation to properties that are monitorable under ideal conditions. However, the original FR/FS classes do not precisely define properties with mutation immunity in many cases. We must define smaller property classes within the framework to identify the properties with trustworthy verdicts over channels with the mutations from Definitions 24-27.

Figure 4.3 shows the FR/FS classification with the additional property classes indicated. In the figure, each area is numbered for ease of reference. Each area may represent multiple classes (if they overlap) and each class may be include multiple areas. For example, Liveness (`NFR`) Properties are represented in the figure by areas 1, 7, 8, 12, 13, and 16. Inclusion Properties, on the other hand, are represented in only area 12.

Proximate Properties

Proximate Properties, which we denote `Prox`, are properties where the duplication of a symbol may change whether or not a trace satisfies or violates the property. For example, $L[\![Xp]\!]$ is Proximate, since the trace $\langle\{\neg p\}, \{p\}, \cdots\rangle$ satisfies the property but $\langle\{\neg p\}, \{\neg p\}, \{p\}, \cdots\rangle$ does not. The intuition behind the name "Proximate" is that these properties depend, in some way, on the proximity of two parts of the trace. In Figure 4.3, areas 9, 10, 14, and 15 contain only Proximate Properties, and areas 5, 6, 8, and 13 *include* Proximate Properties but not *only* Proximate Properties.

Proximate Properties are related to the dual of a class usually called *closed under stuttering* [207], or *stutter-invariant* [179]. Stutter-invariant Properties are those in which any satisfying trace still satisfies the property when symbols are repeated. Proximate is not exactly the dual of stutter-invariant, as Proximate Properties are affected only by *finite* stuttering. This includes most, but not all, LTL formulae that contain the *next* (X) operator. For example, $L[\![GF(p \wedge Xq)]\!]$ is not Proximate because finite duplication of symbols cannot cause a satisfying trace to violate the property. Note that the presence of *next* (X) in an LTL formula is not sufficient to prove inclusion in `Prox` but the absence of X guarantees that the formula is out of `Prox`.

Definition 33 (Proximate Properties). *A given property $\mathcal{L} \subseteq \Sigma^\omega$ is a* Proximate *Property ($\mathcal{L} \in$ `Prox`) iff $\exists \alpha \in \Sigma^*.\ \exists \mu \in \Sigma^\omega.\ \exists x \in \Sigma$ such that either $\alpha \cdot \langle x \rangle \cdot \mu \in \mathcal{L}$, and $\alpha \cdot \langle x, x\rangle \cdot \mu \notin \mathcal{L}$, or $\alpha \cdot \langle x \rangle \cdot \mu \notin \mathcal{L}$, and $\alpha \cdot \langle x, x \rangle \cdot \mu \in \mathcal{L}$.*

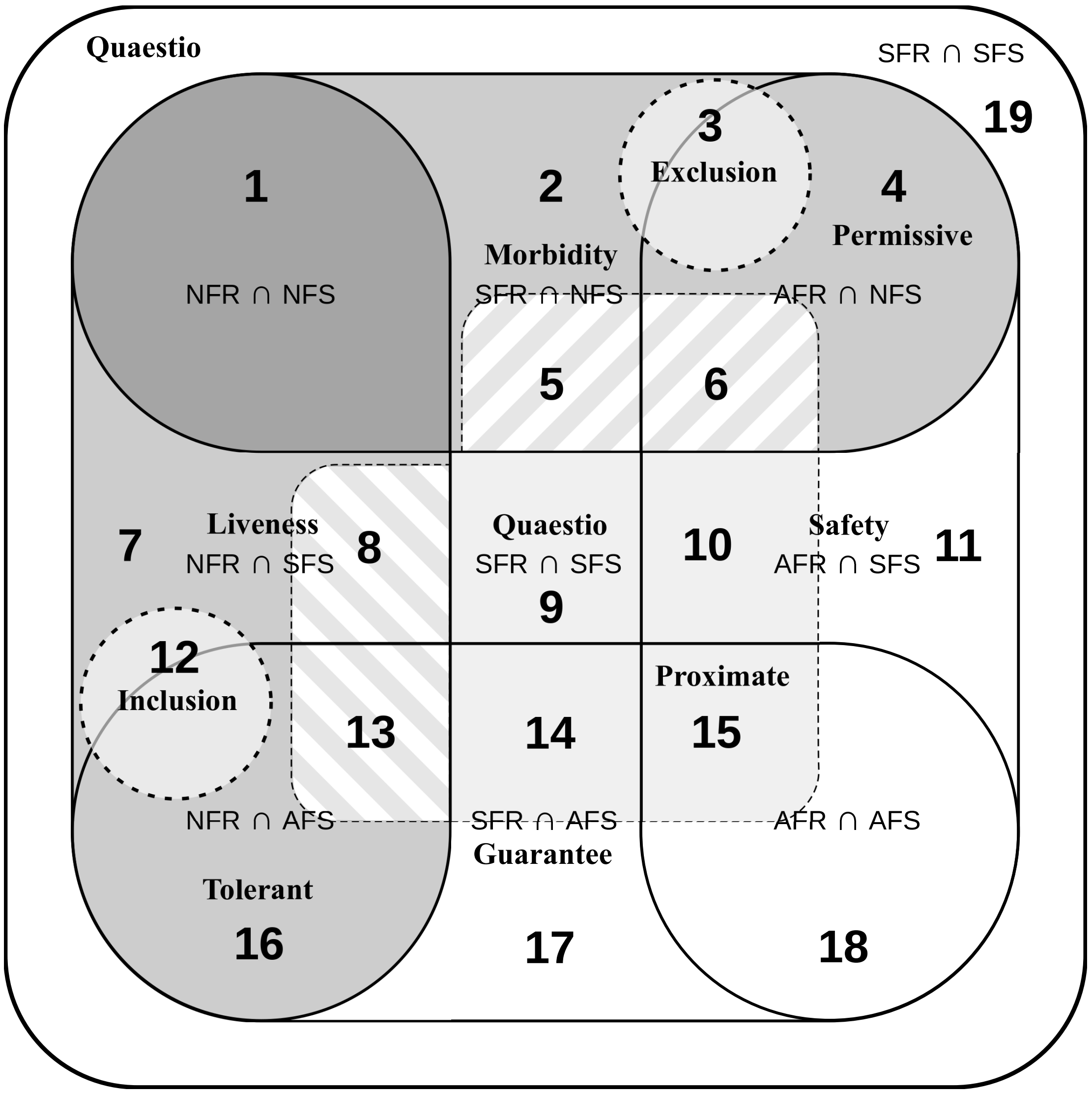


Figure 4.3: FR/RS property classification including Proximate (9, 10, 14, 15, and parts of 5, 6, 8, 13), Tolerant (1, 7, 12, 16), Permissive (1, 2, 3, 4), Inclusion (12), and Exclusion (3) classes

Tolerant Properties

Tolerant Properties, which we denote by the abbreviation `Tolr`, are properties where satisfying traces will still satisfy the property with any finite string inserted into the trace. For example, $L[\![Fp]\!]$ is a Tolerant Property because adding any finite string to a satisfying trace (say, $\langle\{p\},\cdots\rangle$) cannot cause the trace to violate the property. The intuition behind the name "Tolerant" is that the properties tolerate the insertion of a finite string. Tolerant Properties are shown in Figure 4.3 as areas 1, 7, 12, and 16.

Definition 34 (Tolerant Properties). *A given property $\mathcal{L} \subseteq \Sigma^\omega$ is a* Tolerant *Property ($\mathcal{L} \in$ `Tolr`) iff* $\forall \alpha, \beta \in \Sigma^*.\ \forall \mu \in \Sigma^\omega.\ (\alpha \cdot \mu \in \mathcal{L}) \rightarrow (\alpha \cdot \beta \cdot \mu \in \mathcal{L})$.

Tolerant is a subclass of Liveness and is disjoint from Proximate. That Tolerant is disjoint from Proximate is obvious, and we show that it is a subclass of Liveness in Theorem 4. Areas 8 and 13 in Figure 4.3 represent Liveness Properties that are not Tolerant. A conseqence of the differences between Definitions 33 and 34 is that $\texttt{NFR} \cap \texttt{Prox} \subset \texttt{NFR} \setminus \texttt{Tolr}$, however. An example of an `NFR` property that is not Tolerant, but also is not Proximate is $L[\![F(p \wedge Xq)]\!]$. A Proximate Property that is not Tolerant is $L[\![F(p \wedge Xq \wedge XXp)]\!]$. An exact characterization of the properties $\texttt{NFR} \setminus (\texttt{Prox} \cup \texttt{Tolr})$ is unknown and left for future work.

Theorem 4 (Tolerant is a Subclass of Liveness). $\mathit{Tolr} \subset \mathit{NFR}$

Proof: We must consider two cases: if the property is infinitely satisfied, or finitely satisfied.

1. Case 1 (infinite satisfaction): In that case the infinite suffix of the trace $\mu \in \Sigma^\omega$ determines that $\alpha \cdot \mu \in \mathcal{L}$ for any $\alpha \in \Sigma^*$. This is what Sistla called an *absolute liveness* property which are a subset of liveness properties [207].

2. Case 2 (finite satisfaction): Suppose, a property $\mathcal{L}_2 \subseteq \Sigma^\omega$ where the finite prefix determines satisfaction. For clarity, we separate this finite portion into two parts $\alpha, \beta \in \Sigma^*$ such that $\forall \mu \in \Sigma^\omega.\ \alpha \cdot \beta \cdot \mu \in \mathcal{L}_2$. We will prove by contradiction. Now assume there exists a finite trace $\gamma \in \Sigma^*$ and an infinite suffix $\mu_f \in \Sigma^\omega$ such that $\alpha \cdot \gamma \cdot \beta \cdot \mu_f \notin \mathcal{L}_2$. If $\forall \mu_f \in \Sigma^\omega.\ \alpha \cdot \gamma \cdot \beta \cdot \mu_f \notin \mathcal{L}_2$, then $\mathcal{L}_2$ is finitely refutable and not a Liveness property. Otherwise, $\exists \mu_t \in \Sigma^\omega$ such that $\alpha \cdot \gamma \cdot \beta \cdot \mu_t \in \mathcal{L}_2$. If that is true, then it must be that $\exists \mu \in \Sigma^\omega.\ \forall \sigma \in \Sigma^*.\ \sigma \cdot \mu \in \mathcal{L}_2$ which is the definition of a Liveness property. □

Permissive Properties

Permissive Properties, which we denote by the abbreviation `Perm`, are properties where violating traces will still violate the property with any finite string inserted into the trace. For example, $L[\![Gp]\!]$ is a Permissive Property because adding any finite string to a violating trace (say, $\langle\{\neg p\}, \cdots\rangle$) cannot cause the trace to satisfy the property. The intuition behind the name "Permissive" is that the properties permit the insertion of a string (like tolerant, but negative). Permissive Properties are shown in Figure 4.3 as areas 1, 2, 3, and 4.

Definition 35 (Permissive Properties). *A given property $\mathcal{L} \subseteq \Sigma^\omega$ is an* Permissive *Property ($\mathcal{L} \in$ `Perm`) iff* $\forall \alpha, \beta \in \Sigma^*$ **.** $\forall \mu \in \Sigma^\omega$ **.** $(\alpha \cdot \mu \notin \mathcal{L}) \rightarrow (\alpha \cdot \beta \cdot \mu \notin \mathcal{L})$.

Permissive is a subclass of Morbidity and is disjoint from Proximate. That Permissive is disjoint from Proximate is obvious, and we show that it is a subclass of Morbidity in Theorem 5. Areas 5 and 6 in Figure 4.3 represent Morbidity Properties that are not Permissive. A conseqence of the differences between Definitions 33 and 35 is that $\texttt{NFS} \cap \texttt{Prox} \subset \texttt{NFS} \setminus \texttt{Perm}$, however. An example of an `NFS` property that is not Permissive, but also is not Proximate is $L[\![G(p\ Uq)]\!]$. A Proximate Property that is not Permissive is $L[\![G(p \rightarrow Xq)]\!]$. Like with Tolerant and Proximate, an exact characterization of the properties $\texttt{NFR} \setminus (\texttt{Prox} \cup \texttt{Perm})$ is unknown and left for future work.

Theorem 5 (Permissive is a Subclass of Morbidity). $\texttt{Perm} \subset \texttt{NFS}$

Proof: The proof is equivalent to that for Theorem 4 but for Morbidity instead of Liveness. □

Inclusion Properties

Inclusion Properties, which we denote by the abbreviation `Incl`, are always satisfied by the presence of a finite set of symbols. For example, $L[\![Fp]\!]$ is an Inclusion Property because its satisfaction depends only on the presence of one symbol where p holds. Intuitively, Inclusion Properties are restricted to those that can be expressed as LTL formulae of the form Fp where p is propositional, or disjunctions of Inclusion Property formulae with Fp or Gq where p and q are propositional. Inclusion Properties are shown in Figure 4.3 as area 12.

Definition 36 (Inclusion Properties). *A given property $\mathcal{L} \subseteq \Sigma^\omega$ is an* Inclusion *Property ($\mathcal{L} \in$ `Incl`) iff there exists a finite set of symbols $S \subseteq \Sigma$ such that* $\forall \sigma \in \Sigma^\omega$ **.** $(\sigma \in \mathcal{L} \leftrightarrow \forall s \in S$ **.** $s \in \sigma)$.

Theorem 6 (Inclusion is a Subclass of Tolerant and Disjoint from Morbidity).
$\mathtt{Incl} \subset \mathtt{Tolr} \setminus \mathtt{NFS}$

Proof: Clearly, $\mathcal{L} \in \mathtt{Tolr}\ \forall \mathcal{L} \in \mathtt{Incl}$. Given a property $\mathcal{L} \in \mathtt{Incl}$, any trace $\sigma \in \mathcal{L}$ will still satisfy the property with additional symbols. It is also obvious that $\mathcal{L} \notin \mathtt{NFS}\ \forall \mathcal{L} \in \mathtt{Incl}$, since it must be possible to satisfy $\mathcal{L}$ by the inclusion of a finite set of symbols. □

Exclusion Properties

Exclusion Properties, which we denote by the abbreviation Excl, are always violated by the presence of a finite set of symbols. For example, $L[\![G(\neg p)]\!]$ is an Exclusion Property because its satisfaction depends only on the absence any state where p holds. Intuitively, Exclusion Properties are restricted to those that can be expressed as LTL formulae of the form Gp where p is propositional, or conjunctions of Exclusion Property formulae with Fp or Gq where p and q are propositional. Exclusion Properties are shown in Figure 4.3 as area 3.

Definition 37 (Exclusion Properties). *A given property* $\mathcal{L} \subseteq \Sigma^\omega$ *is an* Exclusion *Property* ($\mathcal{L} \in \mathtt{Excl}$) *iff there exists a finite set of symbols* $S \subseteq \Sigma$ *such that* $\forall \sigma \in \Sigma^\omega .\ (\sigma \notin \mathcal{L} \leftrightarrow \forall s \in S .\ s \in \sigma)$.

Theorem 7 (Exclusion is a Subclass of Permissive and Disjoint from Liveness).
$\mathtt{Excl} \subset \mathtt{Perm} \setminus \mathtt{NFR}$

Proof: Like for Theorem 6, $\mathcal{L} \in \mathtt{Perm}\ \forall \mathcal{L} \in \mathtt{Excl}$. Given a property $\mathcal{L} \in \mathtt{Excl}$, any trace $\sigma \notin \mathcal{L}$ will still violate the property with additional symbols. It is also obvious that $\mathcal{L} \notin \mathtt{NFR}\ \forall \mathcal{L} \in \mathtt{Excl}$, since it must be possible to violate $\mathcal{L}$ by the inclusion of a finite set of symbols. □

4.8 Classifying Immune Properties

In this section, we classify trustworthy verdicts in the $\mathbb{B}_3$ truth domain for properties monitored over unreliable channels. As shown in Section 4.6, this classification also serves to categorize properties that are immune to the trace mutations from those unreliable channels. To classify properties, we use the augmented FR/FS classification introduced in Section 4.7. We limit our study to the mutations introduced in Section 4.4.

Table 4.1: Trustworthy $\mathbb{B}_3$ verdicts over unreliable channels by property class

	Loss			***Corruption***			***Stutter***			***OutOfOrder***			
Class	$\top$	$\bot$	*?*	$\top$	$\bot$	*?*	$\top$	$\bot$	*?*	$\top$	$\bot$	*?*	**Example**
$\mathtt{SFR} \cap \mathtt{NFS}$	-	✓^{p}	✗	-	✗	✗	-	✓^{x}	✓^{x}	-	✓^{e}	✓^{e}	$Fp \wedge Gq$
$\mathtt{AFR} \cap \mathtt{NFS}$	-	✓^{p}	✗	-	✗	✗	-	✓^{x}	✓^{x}	-	✓^{e}	✓^{e}	Gp
$\mathtt{AFR} \cap \mathtt{SFS}$	✗	✗	✗	✗	✗	✗	✓^{x}	✓^{x}	✓^{x}	✗	✗	✗	$p \vee Gq$
$\mathtt{AFR} \cap \mathtt{AFS}$	✗	✗	✗	✗	✗	✗	✓^{x}	✓^{x}	✓^{x}	✗	✗	✗	p
$\mathtt{SFR} \cap \mathtt{AFS}$	✗	✗	✗	✗	✗	✗	✓^{x}	✓^{x}	✓^{x}	✗	✗	✗	$p \wedge Fq$
$\mathtt{NFR} \cap \mathtt{AFS}$	✓^{t}	-	✗	✗	-	✗	✓^{x}	-	✓^{x}	✓^{i}	-	✓^{i}	Fp
$\mathtt{NFR} \cap \mathtt{SFS}$	✓^{t}	-	✗	✗	-	✗	✓^{x}	-	✓^{x}	✓^{i}	-	✓^{i}	$Gp \vee Fq$
$\mathtt{NFR} \cap \mathtt{NFS}$	-	-	✓	-	-	✓	-	-	✓	-	-	✓	GFp
$\mathtt{SFR} \cap \mathtt{SFS}$	✗	✗	✗	✗	✗	✗	✓^{x}	✓^{x}	✓^{x}	✗	✗	✗	$(p \vee GFp) \wedge q$

Table 4.1 shows trustworthy verdicts in $\mathbb{B}_3$ for each FR/FS class of properties and each of the four mutations from Section 4.4. In the table, a ✓ indicates that the verdict is trustworthy, a ✗ means that the verdict is not trustworthy, and a - denotes that the verdict is not possible for the given property class. The table also includes an example property for each class. For example, the first row in Table 4.1 shows the results for the $\mathtt{SFR} \cap \mathtt{NFS}$ class, an example of which is the LTL property $Fp \wedge Gq$. The leftmost three cells show the results for the *Loss* mutation. The cells show that the $\top$ verdict is not possible for $\mathtt{SFR} \cap \mathtt{NFS}$ Properties, the $\bot$ verdict is trustworthy (for the Permissive subclass), and the *?* verdict is not trustworthy.

Most of the property classes for which verdicts are trustworthy are subclasses of the original FR/FS classes defined in Section 4.7.2. For these, we annotate the ✓ mark to indicate the precise subclass. A ✓^{p} indicates the verdict is trustworthy for only Permissive properties. A ✓^{t} indicates the verdict is trustworthy for only Tolerant properties. A ✓^{i} denotes that the verdict is trustworthy for only Inclusive Properties. A ✓^{e} denotes that the verdict is trustworthy for only Exclusive Properties. A ✓^{x} indicates the verdict is trustworthy for the given property class *excluding* Proximate Properties.

4.8.1 Channels with Loss

Only the unmonitorable class $\mathtt{NFR} \cap \mathtt{NFS}$ is immune to Loss, but *true* and *false* verdicts are trustworthy over certain properties. *Loss* is interesting because it is the only mutation from Definitions 24-27 for which some properties have both trustworthy and non-trustworthy verdicts.

Theorem 8 (Over Channels with *Loss*, True is Trustworthy only for Tolerant Properties)**.** *Given a property $\mathcal{L}_1 \in$ `Tolr` and a property $\mathcal{L}_2 \subseteq (\Sigma^\omega \setminus$ `Tolr`$)$, for all pairs $(\sigma_1, \sigma_1') \in Loss^1$, $(\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_1)(\sigma_1') = \top) \rightarrow (\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_1)(\sigma_1) = \top)$ and there exists a pair $(\sigma_2, \sigma_2') \in Loss^1$ such that $(\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_2)(\sigma_2') = \top) \wedge (\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_2)(\sigma_2) \neq \top)$.*

Proof: We will show that *true* is trustworthy for exactly the properties `Tolr`. We must show that $\forall \mathcal{L} \in$ `Tolr` there cannot exist a pair of traces $(\sigma, \sigma') \in Loss^1$ such that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = \top \wedge \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma) \neq \top$. Or, to restate using Definition 18, $\sigma' \cdot \mu_t \in \mathcal{L}$ for all infinite suffixes $\mu_t \in \Sigma^\omega$ and there exists an infinite suffix $\mu_f \in \Sigma^\omega$ such that $\sigma \cdot \mu_f \notin \mathcal{L}$. We will prove by contradiction.

From Definition 24, since σ and σ' must not be equal (or they must result in the same verdict), there exist finite traces $\alpha, \beta \in \Sigma^*$ and a symbol $\exists x \in \Sigma$ such that $\sigma = \alpha \cdot \langle x \rangle \cdot \beta$ and $\sigma' = \alpha \cdot \beta$. So, we assume there exist a pair of finite traces $(\alpha \cdot \langle x \rangle \cdot \beta, \alpha \cdot \beta) \in Loss^1$ such that, for all infinite suffixes $\mu_t \in \Sigma^\omega$, $\alpha \cdot \beta \cdot \mu_t \in \mathcal{L}$ and there exists an infinite suffix $\mu_f \in \Sigma^\omega$ such that $\alpha \cdot \langle x \rangle \cdot \beta \cdot \mu_f \notin \mathcal{L}$. For this to be true, it must be that there exists an infinite suffix $\mu \in \Sigma^\omega$ such that $\alpha \cdot \beta \cdot \mu \in \mathcal{L}$ and $\alpha \cdot \langle x \rangle \cdot \beta \cdot \mu \notin \mathcal{L}$. Since $\beta \cdot \mu$ appears in both traces, we can simplify to say that there exists an infinite suffix $\mu \in \Sigma^\omega$ such that $\alpha \cdot \mu \in \mathcal{L}$ and $\alpha \cdot \langle x \rangle \cdot \mu \notin \mathcal{L}$. However, this is the complement of Tolerant Properties from Definition 34, which we have explicitly excluded. □

Theorem 9 (False is Trustworthy only for Permissive Properties over Channels with *Loss*)**.** *Given a property $\mathcal{L}_1 \in$ `Perm` and a property $\mathcal{L}_2 \subseteq (\Sigma^\omega \setminus$ `Perm`$)$, for all pairs $(\sigma_1, \sigma_1') \in Loss^1$, $(\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_1)(\sigma_1') = \bot) \rightarrow (\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_1)(\sigma_1) = \bot)$ and there exists a pair $(\sigma_2, \sigma_2') \in Loss^1$ such that $(\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_2)(\sigma_2') = \bot) \wedge (\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L}_2)(\sigma_2) \neq \bot)$.*

Proof: The proof is identical to that for Theorem 9, but for *false* and Permissive Properties. □

Theorem 10 (`NFR` $\cap$ `NFS` Properties are Vacuously Immune to All Mutations)**.** *Given a property $\mathcal{L} \in$ `NFR` $\cap$ `NFS`, for any finite traces $\sigma, \sigma' \in \Sigma^*$ it is always true that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma)$.*

Proof: The proof is trivial, since $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma) =$ *?* for any finite trace $\sigma \in \Sigma^*$. □

4.8.2 Channels with Corruption

Corruption is the only trace mutation we examine for which no properties, apart from those in $\texttt{NFR} \cap \texttt{NFS}$ have trustworthy verdicts. This result is not surprising, since corruption can change any symbol in the alphabet to any other symbol. *Corruption* cannot change the length of the original trace, so we first show that this does not limit the properties for which the mutation may affect the monitoring verdict.

Lemma 1 (Strings of the Same Length Must Be Able to Result in Different Verdicts)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\omega$, if there exist two finite strings $s, s' \in \Sigma^*$ such that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') \neq \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s)$, then there must exist two finite strings of the same length $\sigma, \sigma' \in \Sigma^* \centerdot |\sigma| = |\sigma'|$ such that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma') \neq \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma)$.*

Proof: First, suppose a property $\mathcal{L} \in \Sigma^\omega$ such that there exist finite traces $s, s' \in \Sigma^*$ such that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') \neq \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s)$, and for all pairs of finite traces of equal length $(\sigma, \sigma') \in \Sigma^* \centerdot |\sigma| = |\sigma'|$, $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = \mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma)$.

If all traces of the same length yield the same verdict, then one of s or s' must be longer than the other (which one does not matter). Assume $|s| > |s'|$. There are three cases:

1. If $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') = \top$ then, from Definition 18, $s' \cdot \mu \in \mathcal{L}$ for all infinite suffixes $\mu \in \Sigma^\omega$. However, for all traces of the same length $t \in \Sigma^* \centerdot |t| = |s'|$, we assume that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(t) = \top$, so there must be a prefix of s where the verdict is $\top$, but this is a contradiction.

2. The same logic applies if $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') = \bot$.

3. If $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') = \textit{?}$, then either $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s) = \top$ or $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s) = \bot$. Suppose that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s) = \top$, as the same argument applies for both verdicts. Then for all finite suffixes $t \in \Sigma^*$ such that s' concatenated with t is the same length as s, $|s' \cdot t| = |s|$, it must be that $(s' \cdot t) \in \mathcal{L}$. However, by Definition 18, there exists an infinite suffix $\mu \in \Sigma^\omega$ such that $(s' \cdot \mu) \notin \mathcal{L}$. Then, there must be either a finite suffix the same length as t, $\sigma \in \Sigma^* \centerdot |s' \cdot \sigma| = |s|$ where $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s' \cdot \sigma) = \bot$ or $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s' \cdot \sigma) = \textit{?}$, which is a contradiction. □

Theorem 11 (If Multiple Verdicts are Possible for a Property, Then None are Trustworthy Over Channels with *Corruption*)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\omega$, for all verdicts $v \in \mathbb{B}_3$, if there exist two finite traces $s, s' \in \Sigma^*$ such that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s') = v$ and $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(s) \neq v$, then there exists a pair of traces $(\sigma, \sigma') \in Corruption^1$ such that $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = v$ and $\mathfrak{E}_{\mathbb{B}_3}(\mathcal{L})(\sigma) \neq v$.*

Proof: Suppose two finite traces with different verdicts $s, s' \in \Sigma^*$. $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(s') = v \wedge \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(s) \neq v$. From Lemma 1 it must be possible for $|s| = |s'|$. Clearly, from Definitions 25 and 28, there exists a number $k \in \mathbb{N}$ such that $(s, s') \in \mathit{Corruption}^k$, since any finite string may appear on the left side of the pair and applying *Corruption* an arbitrary number of times can transform a string to any other string of the same length. From Corollary 1, a verdict is trustworthy for $\mathit{Corruption}^1$ iff it is trustworthy for $\mathit{Corruption}^k$. □

4.8.3 Channels with Stutter

Many works on temporal logic have examined the effects of stuttering. Lamport argued for the omission of the *next* (X) operator in temporal logic, and demonstrated that traces with repeating symbols could not be differentiated without it [142]. The difference between prior work on stuttering and ours is that Definition 26 includes only finite stuttering, while other works have allowed for infinite repetition of a symbol [13, 207, 179]. This difference has significant consequences for what properties are immune to the mutation.

Theorem 12 (All Non-Proximate Properties are Immune to *Stutter*)**.** *Given a property $\mathcal{L} \subseteq (\Sigma^\omega \setminus \texttt{Prox})$, for all pairs $(\sigma, \sigma') \in \mathit{Stutter}^1$, it must be that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma)$.*

Proof: The proof follows directly from Definitions 26 and 33. For all non-Proximate Properties, $\mathcal{L} \in (\Sigma^\omega \setminus \texttt{Prox})$ and for all finite prefixes $\alpha \in \Sigma^*$ for all infinite suffixes $\forall \beta \in \Sigma^\omega$ and for all symbols $x \in \Sigma$, either $\alpha \cdot \langle x \rangle \cdot \beta \in \mathcal{L}$, and $\alpha \cdot \langle x, x \rangle \cdot \beta \in \mathcal{L}$, or $\alpha \cdot \langle x \rangle \cdot \beta \notin \mathcal{L}$, and $\alpha \cdot \langle x, x \rangle \cdot \beta \notin \mathcal{L}$. Clearly, *true* is trustworthy, since for all pairs $(\sigma, \sigma') \in \mathit{Stutter}^1$, if $\sigma' \cdot \mu \in \mathcal{L}$ for all infinite suffixes $\mu \in \Sigma^\omega$, then $\sigma \cdot \mu \in \mathcal{L}$. By the same logic, *false* is trustworthy. If there exist infinite suffixes $\mu_t, \mu_f \in \Sigma^\omega$ such that $\sigma' \cdot \mu_t \in \mathcal{L}$ and $\sigma' \cdot \mu_f \notin \mathcal{L}$, then $\sigma \cdot \mu_t \in \mathcal{L}$ and $\sigma \cdot \mu_f \notin \mathcal{L}$, so *?* is also trustworthy. By Theorem 3 such a property is immune. □

4.8.4 Channels with Out-of-Order

Properties that are immune to *OutOfOrder* are limited to subclasses of Liveness and Morbidity. The Inclusion and Exclusion classes defined in Section 4.7 are limited to properties where satisfaction or violation depend on the presence of specific symbols.

Theorem 13 (Inclusion and Exclusion Properties are Immune to *OutOfOrder*)**.** *Given a property $\mathcal{L} \in \texttt{Incl} \cup \texttt{Excl}$, for all pairs of finite traces $(\sigma, \sigma') \in \mathit{OutOfOrder}^1$, it must be true that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma')$.*

Proof: The proof follows directly from Definitions 27, 36 and 37. By Definition 36, given a property $\mathcal{L} \in \texttt{Incl}$, for all traces in that property $s \in \mathcal{L}$ there exists a set of symbols $X \subseteq \Sigma$ such that for an infinite trace $\sigma \in \Sigma^\omega$, $\sigma \in \mathcal{L}$ iff all of the symbols in X are in σ. By Definition 27, for all pairs of finite traces $(\sigma, \sigma') \in \mathit{OutOfOrder}^1$ there cannot exist a symbol $x \in \Sigma$ such that $x \in \sigma'$ and $x \notin \sigma$. Since all pairs $(\sigma, \sigma') \in \mathit{OutOfOrder}^1$ must contain the same symbols, they must result in the same verdicts. The same logic applies for Definition 37 and violation, rather than satisfaction, of the property. $\square$

Theorem 14 (No Verdicts are Trustworthy for Non-Inclusion, Non-Exclusion Properties Over Channels with *OutOfOrder*)**.** *Given a property $\mathcal{L} \subseteq (\Sigma^\omega \setminus (\texttt{Incl} \cup \texttt{Excl}))$, for all verdicts $v \in \mathbb{B}_3$ there exists a pair of traces $(\sigma, \sigma') \in \mathit{OutOfOrder}^1$ such that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma') = v$ and $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma) \neq v$, except in the case where $\mathcal{L} \in \texttt{NFR} \cap \texttt{NFS}$ and $v = ?$, since that is the only possible verdict for such properties.*

Proof: The proof, again, follows directly from Definitions 27, 36 and 37. Consider a property $\mathcal{L} \subseteq (\Sigma^\omega \setminus (\texttt{Incl} \cup \texttt{Excl}))$. Then, there must exist two infinite traces $\sigma, \sigma' \in \Sigma^\omega$ such that $\sigma \in \mathcal{L}$ and $\sigma' \notin \mathcal{L}$ where all symbols $s \in \Sigma$ occur in both string $s \in \sigma$ and $s \in \sigma'$. In that case, there are two possibilities.

1. Satisfaction or violation depend on infinite strings. In that case, either both satisfaction and violation depend on infinite strings, so $\mathcal{L} \in \texttt{NFR} \cap \texttt{NFS}$ and the verdict is always *?*, or only one depends on infinite strings and the other is covered by the second case.

2. Satisfaction or violation depend on symbol order. In that case, by Definition 27, there exists a pair of finite traces $(\sigma, \sigma') \in \mathit{OutOfOrder}^1$ such that $\sigma \in \mathcal{L}$ and $\sigma' \notin \mathcal{L}$. $\square$

4.8.5 Utility of Mutation Immune Properties

Many properties that are immune to the *Stutter* and *OutOfOrder* mutations or have trustworthy verdicts in the presence of *Loss* are useful. To show the importance of these properties, we provide a classification of property specification patterns from Dwyer, Avrunin, and Corbett's survey [78]. This analysis shows that the most common patterns are monitorable over some unreliable channels.

Table 4.2 shows the property specification patterns from [78] and where they fit in the updated FR/FS classification. Note that we only list patterns in the global scope as these patterns account for 78.9% of all the properties in the survey. In the table, the Pattern

Table 4.2: Property specification patterns

Pattern	**Class**	**Occurence**
Absence	$\texttt{AFR} \cap \texttt{NFS} \cap \texttt{Excl}$	9.4%
Universality	$\texttt{AFR} \cap \texttt{NFS} \cap \texttt{Excl}$	25.1%
Existence	$\texttt{NFR} \cap \texttt{AFS} \cap \texttt{Incl}$	2.7%
Bnd. Existence	$\texttt{AFR} \cap \texttt{NFS} \cap \texttt{Excl}$	0%
Precedence	$\texttt{AFR} \cap \texttt{SFS} \setminus \texttt{Prox}$	55%
Response	$\texttt{NFR} \cap \texttt{NFS}$	5.7%
Precedence Chn.	$\texttt{SFR} \cap \texttt{SFS} \setminus \texttt{Prox}$	1.8%
Response Chn.	$\texttt{NFR} \cap \texttt{NFS}$	0.2%

column gives the name of the pattern, Class gives the classification of that pattern in the updated FR/FS taxonomy, and Occurence gives the incidence of that pattern in the global scope in the original study [78].

All of the patterns in Table 4.2 are immune to at least *Stutter*, and most are immune to or have trustworthy verdicts over other mutations. The Absence, Universality, Existence, and Bounded Existence patterns are all either Inclusive or Exclusive Properties. These patterns are immune to *Stutter* and *OutOfOrder* and have trustworthy verdicts over *Loss* and make up 37.2% of the global-scope properties from [78]. The Precedence and Precedence Chain patterns, which make up 56.8% of global-scope properties, are non-Proximate and immune to *Stutter*. The Response and Response Chain patterns, which only make up 5.9% of global-scope properties, are in $\texttt{NFR} \cap \texttt{NFS}$, which means they are non-monitorable and trivially immune to all mutations.

The property classification in this section is valuable for quickly identifying properties that can be monitored over channels with the *Loss*, *Corruption*, *Stutter*, and *OutOfOrder* mutations. However, custom mutations that more precisely model an unreliable channel must be analyzed separately. This requires a decision procedure that can accommodate any mutation. By Rice's Theorem, monitorability over unreliable channels is undecidable in the general case where the language may require a Turing Machine to express. Most properties of interest, however, including those expressible as LTL, are ω-regular. We now provide a decision procedure for those properties expressible by an NBA.

4.9 Deciding Immunity for Omega-Regular Properties

To determine the immunity of an ω-regular property to a trace mutation, we must construct automata that capture the notion of uncertainty from $\mathbb{B}_3$. Bauer et al. defined a simple process to build a $\mathbb{B}_3$ monitor using two DFAs in their work on LTL$_3$ [30]. We will examine these DFAs to decide if the property is true-false immune to the trace mutation.

Two DFAs are needed to represent the $\mathbb{B}_3$ output of the monitor, since each DFA can only accept or reject a trace. In the monitor, if one DFA rejects the trace then the verdict is $\perp$, if the other rejects the trace then the verdict is $\top$ and if neither reject then the verdict is *?*. It is not possible for both DFAs to reject due to how they are constructed.

The construction procedure for the monitor begins by complementing the property. A language of infinite words $\mathcal{L}$ is represented as an NBA $\mathcal{A}_{\mathcal{L}} = (Q, \Sigma, q_0, \delta, F_{\mathcal{L}})$, for example, an LTL formula can be converted to an NBA by tableau construction [206]. The NBA is then complemented to form $\overline{\mathcal{A}_{\mathcal{L}}} = (\overline{Q}, \Sigma, \overline{q_0}, \overline{\delta}, \overline{F_{\mathcal{L}}})$. *Remark:* The upper bound for NBA complementation is $2^{O(n \log n)}$, so it is cheaper to complement an LTL property and construct its NBA if starting from temporal logic [141].

To form the monitor, create two NFAs based on the NBAs and then convert them to DFAs. The two NFAs are defined as $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ and $\overline{\mathcal{A}} = (\overline{Q}, \Sigma, \overline{q_0}, \overline{\delta}, \overline{F})$ The new accepting states are the states from which an NBA accepting state is reachable. That is, we populate the accepting states so that $F = \{q \in Q : (\mathcal{R}each(\mathcal{A}_{\mathcal{L}}, q) \cap F_{\mathcal{L}}) \neq \varnothing\}$, and $\overline{F} = \{q \in \overline{Q} : (\mathcal{R}each(\overline{\mathcal{A}_{\mathcal{L}}}, q) \cap \overline{F_{\mathcal{L}}}) \neq \varnothing\}$. The two NFAs are then converted to DFAs via subset construction. The verdict for a finite trace σ is then given as the following function $\mathfrak{v}_{\mathbb{B}_3} : 2^{\Sigma^\omega} \to \Sigma^* \to \mathbb{B}_3$.

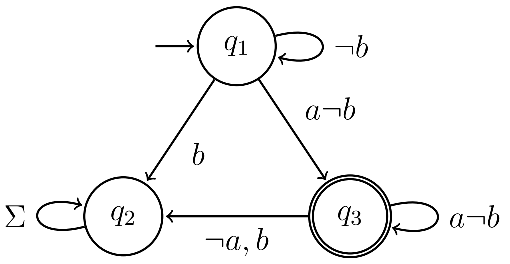
Definition 38 ($\mathbb{B}_3$ Monitor Verdict)**.** *Given a property $\mathcal{L} \subseteq \Sigma^\omega$, derive $\mathbb{B}_3$ monitor DFAs $\mathcal{A}$ and $\overline{\mathcal{A}}$. The $\mathbb{B}_3$ verdict for a string $\sigma \in \Sigma^*$ is the following.*

$$\mathfrak{v}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = \begin{cases} \perp & \text{if } \sigma \notin L(\mathcal{A}) \\ \top & \text{if } \sigma \notin L(\overline{\mathcal{A}}) \\ ? & \text{otherwise} \end{cases}$$

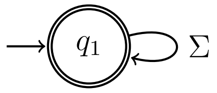
Example: Figure 4.1b shows the NBA $\mathcal{A}_{\mathcal{L}}$ that accepts the infinite-string language of the LTL formula $\varphi = G(a \to F\neg a) \vee Fb$. To construct an LTL$_3$ monitor for φ, we must first complement this NBA, then use the two NBAs to create NFAs and finally DFAs.

Figure 4.4a shows the NBA $\overline{\mathcal{A}_{\mathcal{L}}}$ that accepts the language $L[\![\neg\varphi]\!]$ and is the complement of $\mathcal{A}_{\mathcal{L}}$ in Figure 4.1b. To obtain monitor DFAs, the states and transitions from these

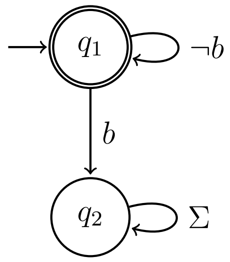
NBAs are used to construct NFAs with new accepting conditions, and then the NFAs are determinized. Figures 4.4b and 4.4c show the simplified monitor DFAs for $L[\![\varphi]\!]$ and $L[\![\neg\varphi]\!]$, respectively. The monitor reaches a $\top$ verdict if the input trace prefix contains a symbol where b holds, otherwise the verdict is *?*.



(a) NBA $\overline{\mathcal{A}_{\mathcal{L}}}$



(b) DFA $\mathcal{A}$



(c) DFA $\overline{\mathcal{A}}$

Figure 4.4: NBA for $L[\![\neg\varphi]\!]$ and its $\mathbb{B}_3$ monitor DFAs

We can now restate Definition 30 using monitor automata. This new definition will allow us to construct a decision procedure for a property's immunity to a mutation.

Theorem 15 (True-False Immunity to Unreliable Channels for ω-Regular Properties)**.** *Given an ω-regular language $\mathcal{L} \subseteq \Sigma^\omega$, derive $\mathbb{B}_3$ monitor DFAs $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ and $\overline{\mathcal{A}} = (\overline{Q}, \Sigma, \overline{q_0}, \overline{\delta}, \overline{F})$. $\mathcal{L}$ is true-false immune to a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$ iff for all pairs of finite traces in the mutation $(\sigma, \sigma') \in \mathcal{M}^k$, it must be that $(\sigma \notin L(\mathcal{A}) \Leftrightarrow \sigma' \notin L(\mathcal{A}))$ and $(\sigma \notin L(\overline{\mathcal{A}}) \Leftrightarrow \sigma' \notin L(\overline{\mathcal{A}}))$.*

Proof: By Definition 30 we need only show that $\mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma) = \mathfrak{L}_{\mathbb{B}_3}(\mathcal{L})(\sigma')$ is equivalent to $(\sigma \notin \mathrm{L}(\mathcal{A}) \Leftrightarrow \sigma' \notin \mathrm{L}(\mathcal{A}))$ and $(\sigma \notin \mathrm{L}(\overline{\mathcal{A}}) \Leftrightarrow \sigma' \notin \mathrm{L}(\overline{\mathcal{A}}))$. There are three cases: $\perp$, $\top$, and *?*. For $\perp$ and $\top$ it is obvious from Definition 38 that the verdicts are derived from exclusion from the languages of $\mathcal{A}$ and $\overline{\mathcal{A}}$. As there are only three possible verdicts, this also shows the *?* case. $\square$

We say that an automaton is immune to a trace mutation in a similar way to how a property is immune. To show that a property is true-false immune to a mutation, we only need to show that its $\mathbb{B}_3$ monitor automata are also immune to the property. Note that, since the implication is both directions, we can use either language inclusion or exclusion in the definition.

Definition 39 (Finite Automaton Immunity)**.** *Given an FA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ and a trace mutation $\mathcal{M}^k \subseteq \Sigma^* \times \Sigma^*$, $\mathcal{A}$ is immune to $\mathcal{M}^k$ iff for all pairs of finite traces in the mutation $(\sigma, \sigma') \in \mathcal{M}^k$, it must be that $\sigma \in L(\mathcal{A}) \Leftrightarrow \sigma' \in L(\mathcal{A})$.*

With this definition we can provide a decision procedure for the monitorability of an ω-regular property over an unreliable channel. The procedure will check the immunity of the $\mathbb{B}_3$ monitor automata to the mutations from the channel, as well as the property's monitorability. If the DFAs are both immune to the mutations and the property is monitorable, then the property is monitorable over the unreliable channel.

4.10 Decision Procedure for DFA Immunity

We propose Algorithm 7 for deciding whether a DFA is immune to a trace mutation. The algorithm is loosely based on Hopcroft and Karp's near-linear algorithm for determining the equivalence of finite automata [114].

Algorithm 7 Determine if a DFA is immune to a given trace mutation.

```
1: procedure IMMUNE( $\mathcal{A} = (\Sigma, Q, q_0, \delta, F), M$ )
2:     for $q \in Q$ do $E(q) \leftarrow \{q\}$                              ▷ $E$ is a map
3:     $R \leftarrow \mathcal{R}each(\mathcal{A}, q_0)$                          ▷ $R$ is the reachable states
4:     $T \leftarrow \{\ \}$                                                    ▷ $T$ is a worklist
5:     for $(\sigma, \sigma') \in M$ where $|\sigma| = minLength(M)$ do
6:         for $q \in R$ do
7:             $q_1 \leftarrow \delta^*(q, \sigma)$                              ▷ Follow original trace
8:             $q_2 \leftarrow \delta^*(q, \sigma')$                             ▷ Follow mutated trace
9:             $E(q_1) \leftarrow E(q_2) \leftarrow \{q_1, q_2\}$
10:            $T \leftarrow T \cup \{(q_1, q_2)\}$
11:    while $T$ is not empty do
12:        let $(q_1, q_2) \in T$                                           ▷ Get a pair from the worklist
13:        $T \leftarrow T \setminus \{(q_1, q_2)\}$                            ▷ Remove the pair from $T$
14:        for $\alpha \in \Sigma$ do
15:            $n_1 \leftarrow \delta(q_1, \alpha)$
16:            $n_2 \leftarrow \delta(q_2, \alpha)$
17:            $C \leftarrow \{E(n_1), E(n_2)\}$
18:            if $|C| > 1$ then
19:                $E(n_1) \leftarrow E(n_2) \leftarrow \bigcup C$                ▷ Merge sets in $E$
20:                $T \leftarrow T \cup \{(n_1, n_2)\}$
21:    if Any set in $E$ contains both final and non-final states then return False
22:    else return True
```

Algorithm 7 checks if the DFA $\mathcal{A}$ is immune to the mutation M, where $\mathcal{A}$ represents part of the $\mathbb{B}_3$ monitor for a property and M is a relation given by $\mathcal{M}^1$ in Definition 28. The intuition behind Algorithm 7 is to follow transitions for pairs of unmutated and corresponding mutated strings in M and verify that they lead to the same acceptance verdicts.

More specifically, Algorithm 7 finds sets of states which must be equivalent for the DFA to be immune to a given mutation. The final verdict of IMMUNE is found by checking that no equivalence class contains both final and non-final states. If an equivalence class contains both, then there are some strings for which the verdict will change due to the given mutation.

If all mutations required only a string of length one, the step at Lines 7 and 8 could follow transitions for pairs of single symbols. However, mutations like *OutOfOrder* require strings of at least two symbols, so we must follow transitions for short strings. We express this idea of a minimum length for a mutation in the $minLength : 2^{\Sigma^* \times \Sigma^*} \rightarrow \mathbb{N}$ function. For mutations in Section 4.4, $minLength(Loss) = minLength(Corruption) = minLength(Stutter) = 1$ and $minLength(OutOfOrder) = 2$. Note that *minLength* for unions must increase to permit the application of both mutations on a string. For example, $minLength(Loss \cup Corruption) = 2$. This length guarantees that each string has at least one mutation, which is sufficient to show immunity by Theorem 2.

The algorithm works as follows. We assume a mutation can occur at any time, so we begin by following transitions for pairs of mutated and unmutated strings from every reachable state (stored in the set R). On Lines 5-10, for each pair (σ, σ') in M and for each reachable state, we compute the states q_1 and q_2 reached from σ (respectively σ'). The map E contains equivalence classes, which we update for q_1 and q_2 to hold the set containing both states. The pair of states is also added to the worklist T, which contains equivalent states from which string suffixes must be explored.

The loop on Lines 11-20 then explores those suffixes. It takes a pair of states (q_1, q_2) from the worklist and follows transitions from those states to reach n_1 and n_2. If n_1 and n_2 are already marked as equivalent to other states in E or aren't marked as equivalent to each other, those states are added to the worklist, and their equivalence classes in E are merged. If at the end, there is an equivalence class with final and non-final states, then $\mathcal{A}$ is not immune to M.

Theorem 16 (Immunity Procedure Correctness)**.** *Algorithm 7 is sound and complete for any DFA and prefix-assured mutation. That is, given a DFA* $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$*, and a mutation,* M*,* $\text{IMMUNE}(\mathcal{A}, M) \Leftrightarrow \mathcal{A}$ *is immune to* M*.*

Proof: By Definition 39, this is equivalent to showing that $\text{IMMUNE}(\mathcal{A}, M) \Leftrightarrow (\forall (\sigma, \sigma') \in M,\ \sigma \in \mathrm{L}(\mathcal{A}) \Leftrightarrow \sigma' \in \mathrm{L}(\mathcal{A}))$.

We will prove the $\Rightarrow$ direction (soundness) by contradiction. Suppose at the completion of the algorithm that all sets in E contain only final or non-final states, but that $\mathcal{A}$ is not immune to M. There is at least one pair $(\sigma_b, \sigma_b') \in M$ where one leads to a final state, and one does not. If Algorithm 7 had checked this pair then these states would be in an equivalence class in E. Since the loop on Line 7 follows transitions for pairs in M of length $minLength(M)$, the reason (σ_b, σ'_b) was not checked must be because $|\sigma_b| \neq minLength(M)$. The length of σ_b must be greater than $minLength(M)$ since strings shorter than $minLength(M)$ cannot be mutated by M. Since M is prefix-assured, there must be a pair $(\sigma, \sigma')\textbf{.}\ |\sigma| = minLength(M)$ that are prefixes of (σ_b, σ'_b). The loop on Line 11 will check $(\sigma \cdot s, \sigma' \cdot s)\ \forall s \in \Sigma^*$. Therefore it must be the case that there exist two different finite suffixes $t, u \in \Sigma^*\textbf{.}\ t \neq u$ such that $\sigma_b = \sigma \cdot t$ and $\sigma'_b = \sigma' \cdot u$. However, if $t \neq u$ then $(\sigma_b, \sigma'_b) \in M^k$ for some $k > 1$, so $\mathcal{A}$ is immune to M^1 but not M^k, but from Theorem 2 this is a contradiction.

We prove the $\Leftarrow$ direction (completeness) by induction. We will show that if $\mathcal{A}$ is immune to M then no set in E, and no pair in T will contain both final and non-final states. The base case at initialization is obviously true since every set in E contains only one state and T is empty. The induction hypothesis is that at a given step i of the algorithm if $\mathcal{A}$ is immune to M then every set in E and every pair in T contains only final or non-final states.

At step $i + 1$, in the loop starting at Line 7, E and T are updated to contain states reached by following σ and σ'. Clearly, if $\mathcal{A}$ is immune to M then these states must be both final or non-final since we followed transitions from reachable states for a pair in M. In the loop on Line 11, n_1 and n_2 are reached by following the same symbol in the alphabet from a pair of states in T. If $\mathcal{A}$ is immune to M, the strings leading to that pair of states must both be in, or both be out of the language. So, extending both strings by the same symbol in the alphabet creates two strings that must both be in or out of the language. These states reached by following these strings are added to T on Line 20.

On Lines 17 and 19, the two sets in E corresponding to n_1 and n_2 are merged. Since both sets must contain only final or non-final states, and one-or-both of n_1 and n_2 are contained in them, the union of the sets must also contain only final or non-final states. □

Theorem 17 (Immunity Procedure Complexity)**.** *Algorithm 7 is Fixed-Parameter Tractable. That is, given a DFA $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, and a mutation, M, its maximum running time is $|Q|^{O(1)} f(k)$, where f is some function that depends only on some parameter k.*

Proof: The run-time complexity of Algorithm 7 is $O(n)O(m^l\ f(M))$ where $n = |Q|$, $m = |\Sigma|$, $l = minLength(M)$, and f is a function on M. First, Lines 4, 7, 8, 9, 10, 12, 13, 15, 16, 17, 18, 19, and 20 execute in constant time, while each of Lines 2, 3, and 21 run in time bounded by n.

The initialization loop at Line 5 runs once for each pair in the mutation where the length of σ is bounded by $\mathit{minLength}(M)$. This count is m^l times a factor $f(M)$ determined by the mutation. For example, $f(\mathit{Loss}) = l$ because each σ is mutated to remove each symbol in the string. Critically, this factor $f(M)$ must be finite, which it is for the mutations $\mathcal{M}^1$. The loop at Line 6 runs in time bounded by n, so the body of the loop is reached at most $m^l f(M)n$ times.

The loop at Line 11 may run at most $m^l f(M) + n$ times. The loop continues while the worklist T is non-empty. Initially, T has $m^l f(M)$ elements. Each time Line 13 runs, an element is removed from the worklist. For an element to be added to T, it must contain states corresponding to sets in E which differ. When this occurs, those two corresponding sets are merged, so the number of unique sets in E is reduced by at least one. Therefore, the maximum number of times Line 20 can be reached and an element added to T is n. $\square$

Note that, in practice, $\mathit{minLength}(M)$ is usually small (often only one), so Algorithm 7 achieves near linear performance in the size of the FA. The size of the alphabet has an effect but it is still quadratic.

4.11 Related Work

Unreliable channels have been acknowledged in formal methods research for some time. For example, Lamport suggested in 1983 that temporal logics without *next* operators were immune to stutter [142]. More recent works by Purandare et al. [187] and Lomuscio et al. [154] applied the principle suggested by Lamport for performance optimizations.

In this section, we describe related work in three areas. First, on works examining unreliable channels in RV, second, on the study of unreliable channels as they relate to communicating finite state machines (CFSMs), and finally, on other definitions of monitorability.

4.11.1 Runtime Verification

RV seeks to decide whether a trace generated by the execution of a program satisfies a specification, often expressed in a temporal logic like LTL [21]. Most RV methods assume an ideal trace, but the topic of unreliable channels is of growing interest in the field.

Work has been done to show which properties are verifiable on a trace with mutations and to express degrees of confidence when they are not. Stoller et al. used HMMs to compute the probability of a property being satisfied on a lossy trace [214]. Their definition of lossy included a "gap" marker indicating where symbols were missing. They used HMMs to predict the missing states where gaps occurred and aided their estimations with a learned probability distribution of state transitions. Joshi et al. introduced an algorithm to determine if a specification could be monitored soundly in the presence of a trace with transient loss, meaning that eventually it contained successfully transmitted events [122]. They defined monotonicity to identify properties for which the verdicts could be relied upon once a decision was made.

Garg et al. introduced a first-order logic with restricted quantifiers for auditing incomplete policy logs [95]. The authors used restricted quantifiers to allow monitoring policies that would, in principle, require iterating over an infinite domain. Basin et al. also specified a first-order logic for auditing incomplete policy logs [28]. Basin et al. also proposed a semantics and monitoring algorithm for Metric Temporal Logic (MTL) with freeze quantifiers that was sound and complete for unordered traces [27]. Their semantics was based on a three-value logic, and the monitoring algorithm was evaluated over ordered and unordered traces. All three of these languages used a three value semantics $(t, f, \perp)$ to model a lossy trace, where $\perp$ represented missing information.

Leuker et al. introduced a technique for a Stream Runtime Verification (SRV) over incomplete traces [148]. They defined an abstract form of the TeSSLa SRV language and showed how it could be used to obtain sound verdicts on traces with well defined gaps. Abstract verdicts were clearly delineated from concrete ones, so that imprecise results could not be confused for incorrect results. Their work assumed that missing values were within a known range and that gaps were identifiable.

Li et al. examined out-of-order data arrival in CEP systems and found that SASE [232] queries processed using the Active Instance Stack (AIS) data structure would fail in several ways [150]. They proposed modifications to AIS to support out-of-order data and found acceptable experimental overhead to their technique.

Baader, Bauer, and Tiu examined the complexity of regular language inclusion and exclusion of a finite trace with lost symbols [17]. They modeled traces as patterns where missing sequences were replaced with variables and considered both the linear case, where variables were unique, and the non-linear case, where they could repeat. The authors showed that, for languages specified as an NFA, linear exclusion was solvable in polynomial time while non-linear exclusion was PSPACE-Complete. For inclusion, they found that both the linear and non-linear cases were PSPACE-Complete.

Runtime verification in the presence of noise has been studied in the context of Analog and Mixed Signal (AMS) components, also referred to as mixed signal circuits. These integrate analog circuits and digital circuits; e.g. such a component can transform an analog signal to a digital signal. Wang et al. describe using runtime verification in combination with Monte Carlo simulation (called statistical runtime verification) to analyze Jitter [227]. Jitter is defined as the deviation in time between a noisy signal and an ideal one. A related concept is the notion of system instability, where control outputs ocillate permanently while inputs are constant. Halbwachs et al. proposed a method to verify the stability of systems using heuristics to check strongly connected components of an operator network [101].

4.11.2 Communicating Finite State Machines

Several works in information theory have modeled the problem of unreliable communication channels in CFSMs [40]. CFSM communication channels are treated as unbounded first-in first-out (FIFO) buffers between finite state machines (FSMs), which is a Turing complete model of computation for a class of infinite-state systems called simple reactive programs [230]. Simple reactive programs are data-independent and are useful for modeling communication protocols like the Alternating Bit Protocol [24] and High-level Data Link Control (HDLC) [119]. CFSMs also form the basis of protocol specification languages such as Estelle [41], and Specification and Description Language (SDL) [34]. CFSMs with unreliable communcations channels are no longer Turing complete, and a number of useful properties have been shown to hold in such cases.

Finkel introduced his notion of completely specified protocols to show that they are a class of machines for which the termination problem is decidable [91]. He defined a completely specified protocol as a CFSM where any FSM can receive any message in any local state and can stay in that state, and he showed that protocols using lossy FIFO channels are examples of such protocols. Abdulla and Jonsson later provided algorithms for deciding the termination problem for protocols on lossy FIFO buffers, as well as algorithms for some safety and eventuality properties [3].

Cécé et al. expanded this examination of unreliable FIFO channels in CFSMs by considering channels with insertion errors, duplication errors, and a combination of insertion, duplication, and lossy errors [43]. Their work defined insertion errors in FIFO buffers to be equal to our general notion of noisy traces, but their duplication errors were restricted to consecutive duplicates. They showed that noisy errors on a communication channel between two FSMs decrease the expressive power of the system more than lossy errors, while consecutive duplication errors do not decrease its expressive power at all.

Iyer and Narasimha introduced probability to the notion of lossy communications channels [120]. They argue that this is a more realistic notion of loss, as hardware reliability statistics are often known. Their work included algorithms for solving probablistic notions of reachability and model checking. That is, given a channel with a known probability of loss, they asked whether a global state in the CFSM was reachable with a certain probability and tolerance, and whether a Propositional Temporal Logic (PTL) property was true with a certain probability and tolerance. Baier and Engelen proved that the set of message sequences on a probabilistic lossy channel that satisfy an LTL property could be decided with probability 1 if the probability of message loss was at least 1/2 [18]. Abdulla et al. proved that, if the probability of message loss was less than 1/2 then the same problem was undecidable [2].

Peng and Makki introduced lossy communicating finite state machines (LCFSMs) to simplify protocol modeling for lossy channels [180]. Traditionally, loss in unreliable communications channels has been modeled using the addition of extra CFSMs which consume messages. The authors argued that this leads to messy CFSM specifications which obfuscate the protocol being modeled. They introduced a delete action to allow the removal of these extra CFSMs.

4.11.3 Other Definitions of Monitorability

Some other definitions of monitorability exist which are outside the scope of this work. These solutions either assume partial knowledge of the monitored system or concern monitoring multiple systems simultaneously.

Sistla, Žefran, and Feng defined monitorability and *strong monitorability* for partially observable stochastic systems modeled as HMMs [208]. Gondi, Patel, and Sistla had already introduced this notion in their work on external monitoring of ω-regular properties of stochastic systems [99], but the later work focused on formalizing the concept and on *internal* monitoring. In these works, properties to be monitored are given as deterministic Streett automata [199] and a model of the system is supplied as a HMM. This varies from definitions of monitoring where only a trace of the output symbols from the monitored system is assumed to be known.

Sistla et al. use *Acceptance Accuracy (AA)* and *Rejection Accuracy (RA)* to define monitorability and strong monitorability, and define them as properties of both a monitored *formula* and a monitored *system*. AA is given as the probability that a monitor accurately returns a positive verdict (accepts) for a formula and a system model, while RA is given as the probability that a monitor accurately returns a negative verdict (rejects) for a formula and a system model. Sistla et al. thus define that a system is strongly monitorable with respect to a formula if there exists a monitor such that both the AA and RA are 1. They then define that a system is monitorable with respect to a formula if there exists a monitor(s) such that accuracies arbitrarily close to 1 may be achieved. The authors conclude that all properties that can be represented as Streett automata are considered externally monitorable for finite state systems and safety properties are also strongly monitorable.

Agrawal and Bonakdarpour first proposed monitoring hyperproperties and introduced a notion of monitorability for such properties [7]. Hyperproperties are sets of sets of traces where monitoring requires reasoning about many prefixes simultaneously. The authors introduce a three-valued semantics for the hyperproperty specification language HYPERLTL [51] and define monitorable classes in that logic. Stucki et al. proposed incorporating partial or complete knowledge of the system into monitoring hyperproperties [215]. They showed that monitoring hyperproperties without such information is infeasible in general and refined Agrawal and Bonakdarpour's definition of hyperproperty monitorability to incorporate computability of the monitor.

Francalanza, Aceto, and Ingolfsdottir defined monitorability for μ-Hennessy-Milner Logic (μHML), a branching time logic for RV based on the modal μ-calculus [93]. They characterized what properties of μHML are monitorable and gave a method to synthesize monitors for those properties. Aceto et al. later introduced a hierarchy of monitorable fragments for μHML and established different guarantees for each fragment [4].

4.12 Conclusions and Future Work

The mutations from Definitions 24 to 27 are useful abstractions of common problems in communication. However, in many cases, they are stronger than is needed as practitioners may have knowledge of the channel that constrains the mutations. For example, on Mars Science Laboratory, messages contain sequence numbers which can be used to narrow the range of missing symbols. Although the property classification from Section 4.8 cannot be used for custom mutations, mutations can be easily defined and then properties can be tested for immunity using Algorithm 7. Custom mutations should avoid behavior that requires long strings to mutate, however, as this causes exponential slowdown. Future work should incorporate a decision procedure for trustworthy verdicts that can be used for custom mutations.

Well designed mutations like those from Definitions 24-27 can be checked quickly. However, the method relies on $\mathbb{B}_3$ monitor construction to obtain DFAs, and the procedure to create them from an NBA is in 2EXPSPACE. We argue that this is an acceptable cost of using the procedure since it is done offline and a monitor must be derived to check the property in any case. Future work should explore ideas from the study of monitorability [68, 178] to find a theoretical bound on deciding immunity and to explore algorithms that do not require monitor construction.

Another avenue for improving our work is to incorporate partial system models to reduce the range of unmutated strings as in grey-box monitoring [215]. Currently, the definition of immunity to a mutation requires that any string (using the alphabet) could be mutated. For many systems, this is more general than is needed, and constraining unmutated strings can allow for more properties to be considered immune and therefore monitorable.

Our definition of monitorability also assumes that every verdict must be trustworthy for a mutation, but some properties may be useful to monitor where only some verdicts are trustworthy. This is similar to how Weak Monitorability relaxes the requirement from Classical Monitorability that every execution may reach a true or false verdict. It may be interesting to define a notion of Weak Monitorability over Unreliable Channels that only requires true and false to be trustworthy.

The ability to check properties expressible by NBAs for monitorability over unreliable channels allows RV to be considered for applications where it would have previously been ignored. To arrive at this capability, we first needed to define monitorability over unreliable channels using both existing notions of monitorability and a new concept of mutation immunity. We proved that immunity to a single application of a mutation is sufficient to show immunity to any number of applications of that mutation, and we defined true-false immunity using $\mathbb{B}_3$ semantics. The FR/FS classification provided a framework that we extended to categorize the properties that are immune to common mutations. In some cases, we found that properties had trustworthy verdicts when monitored over an unreliable channel, despite not being immune to the mutation from that channel.

We believe unreliable communication is an important topic for RV and other fields that rely on remote systems, and we hope that this work leads to further examination of unreliable channels in the RV community.

Chapter 5

Conclusion

For Runtime Verification to gain acceptance as a method to formally verify safety-critical software, practical limitations must be overcome. This thesis introduces improvements to the state-of-the-art that improve the usability of RV in the real world.

Practical barriers to the adoption of Runtime Monitoring occur in every component of an RV system. In Chapter 1, four problems were introduced:

1. that RV methods assume the integrity of execution traces,
2. that monitors are challenging to deploy and may only operate offline,
3. that RV techniques require known formal requirements, and
4. that the verdicts output by monitors are insufficient for error diagnosis and recovery.

`Nfer` is a language and system to infer a hierarchy of temporal interval abstractions from an event stream. `Nfer` provides a formalism based on relative temporal relationships that is simple to adopt for planning software users. It includes the ability to mine abstractions from historical data and offers integrations with popular programming languages. This work addresses problem 3 by supporting a method to mine rules and problem 4 by computing temporal abstractions useful for human and machine comprehension.

Palisade is a framework for distributed online anomaly detection that balances detection latency with ease-of-use. Palisade allows multiple detectors, including `nfer`, to combine to detect a combination of different anomaly symptoms. Palisade is currently in use in several industry collaborations and continues to be extended to support sources, sinks, and processors. This work addresses problem 2 by providing a simple mechanism for distributing online, low-latency monitors.

The definition of monitorability over unreliable channels provides a means to operate traditional RV tools under realistic networking conditions. Properties may be classified as monitorable given prior knowledge of possible trace mutations, even if those mutations are more implementation-specific than the general ones provided. The work provides a platform on which further theories may be built regarding the effects of unreliable communication on runtime monitoring. This work addresses problem 1 by categorizing precisely what program properties may be monitored given inconsistent data.

5.0.1 Future Work

We have made some progress on addressing the barriers to the widespread adoption of RV in safety-critical embedded systems, but much work remains. There are several areas in which we continue to work to improve upon the results in this thesis.

One area of continuing progress is in expanding the usability and effectiveness of `nfer`. The original research was done to support the use case at JPL described in Chapter 2. However, we believe that temporal intervals are a natural abstraction for event sequences, and, as such, we are expanding how `nfer` can be used to understand other types of systems. For example, we are interested in using `nfer` to debug multi-threaded and multi-process Python programs. We can instrument a program to visualize each function execution as an interval, but the question of which functions are the most interesting to examine is unsolved. One solution could be to extend the mining capability to include heuristics specific to the application.

The work in Chapter 4 defines which ω-regular properties are monitorable in the presence of both generic and custom trace mutations. However, more could be done to assist users with crafting custom trace mutations. It may be possible to mine trace mutations from historical data, for example. Another avenue for extending our contribution is to examine property immunity for other language classes. We are interested in the immunity to trace mutations of properties from real-time logics like MTL, from non-propositional logics like Signal Temporal Logic (STL), and from other infinite-trace language classes like ω-context-free languages. We are also interested in the immunity to trace mutations of monitors with richer outputs, such as `nfer`. Some initial work on this problem has been promising.

These contributions represent progress towards practical RV. More work must be done for monitoring to gain acceptance in industry, however. We believe that improving the safety of systems on which lives depend is ultimately worth the effort.

References

[1] Triceps: An innovative CEP. http://triceps.sourceforge.net/. Accessed: 2017-10-18.

[2] Parosh Abdulla, Christel Baier, Purushothaman Iyer, and Bengt Jonsson. Reasoning about probabilistic lossy channel systems. In *Int. Conference on Concurrency Theory (CONCUR'20)*, volume 1877 of *LNCS*, pages 320–333. Springer, 2000.

[3] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[4] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: From branching to linear time and back again. In *Symposium on Principles of Programming Languages (POPL'19)*, volume 3. ACM, January 2019.

[5] Basant Agarwal and Namita Mittal. Hybrid approach for detection of anomaly network traffic using data mining techniques. *Procedia Technology*, 6:996 – 1003, 2012. 2nd International Conference on Communication, Computing & Security [ICCCS-2012].

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, 3 1995.

[7] S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Computer Security Foundations Symposium (CSF'16)*, pages 239–252. IEEE, 2016.

[8] Shikha Agrawal and Jitendra Agrawal. Survey on anomaly detection using data mining techniques. *Procedia Computer Science*, 60:708–713, 2015. Knowledge-Based and Intelligent Information & Engineering Systems 19th Annual Conference, KES-2015, Singapore, September 2015 Proceedings.

[9] Malik Agyemang, Ken Barker, and Rada Alhajj. A comprehensive survey of numeric and symbolic outlier mining techniques. *Intelligent Data Analysis*, 10(6):521–538, 2006.

[10] Aws Albarghouthi, Jorge A Baier, and Sheila A McIlraith. On the use of planning technology for verification. In *Proc. of the ICAPS Workshop on Verification & Validation of Planning & Scheduling Systems (VVPS)*. Citeseer, 2009.

[11] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[12] James F Allen. Towards a general theory of action and time. *Artificial intelligence*, 23(2):123–154, 1984.

[13] Bowen Alpern, Alan J. Demers, and Fred B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, 1986.

[14] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands*, volume 9632 of *LNCS*, pages 15–40. Springer, April 2016.

[15] Embedded trace macrocell architecture specification, 5 2019. Available online: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/.

[16] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 429–435, New York, NY, USA, 2002. ACM.

[17] Franz Baader, Andreas Bauer, and Alwen Tiu. Matching trace patterns with regular policies. In *Int. Conference on Language and Automata Theory and Applications (LATA'09)*, volume 5457 of *LNAI*, pages 105–116. Springer, 2009.

[18] Christel Baier and Bettina Engelen. *Establishing Qualitative Properties for Probabilistic Lossy Channel Systems*, volume 1601 of *LNCS*, pages 34–52. Springer, 1999.

[19] Z. A. Bakar, R. Mohemad, A. Ahmad, and M. M. Deris. A comparative study for outlier detection techniques in data mining. In *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pages 1–6, June 2006.

[20] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *Proc. of the 18th Int. Symposium on Formal Methods (FM'12)*, pages 68–84. Springer, 2012.

[21] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937, pages 44–57, 2004.

[22] Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.

[23] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 20(3):675–706, 2008.

[24] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[25] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. MONPOLY: Monitoring usage-control policies. In *2nd Int. Conference on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 360–364. Springer, 2011.

[26] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 6 2015.

[27] David Basin, Felix Klaedtke, and Eugen Zălinescu. Runtime verification of temporal properties over out-of-order data streams. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 356–376, Cham, 2017. Springer International Publishing.

[28] David A Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. Monitoring compliance policies over incomplete and disagreeing logs. In *RV*, pages 151–167. Springer, 2012.

[29] Sabyasachi Basu and Martin Meckesheimer. Automatic outlier detection for time series: an application to sensor data. *Knowledge and Information Systems*, 11(2):137–154, 2 2007.

[30] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, pages 260–272, Berlin, Heidelberg, 2006. Springer.

[31] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Taşıran, editors, *7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.

[32] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[33] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 9 2011.

[34] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with applications from protocol specification.* Prentice-Hall, Inc., 1991.

[35] Steven M. Bellovin. Packets found on an internet. *SIGCOMM Comput. Commun. Rev.*, 23(3):26–31, 7 1993.

[36] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

[37] Alexander Bergmayr, Hugo Bruneliere, Jordi Cabot, Jokin Garcia, Tanja Mayerhofer, and Manuel Wimmer. fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis. In *Modeling in Software Engineering (MiSE), 2016 IEEE/ACM 8th International Workshop on*, MiSE '16, pages 20–26, New York, NY, United States, 2016. ACM.

[38] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[39] Alexander Bolshev, Jason Larsen, Marina Krotofil, and Reid Wightman. A rising tide: Design exploits in industrial control systems. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, pages 1–11, Austin, TX, 2016. USENIX Association.

[40] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.

[41] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, Mar 1987.

[42] Ricky W Butler, Radu I Siminiceanu, and CA Muno. The ANMLite language and logic for specifying planning problems. Technical report, NASA Langley Research Center; Hampton, VA, United States, 2007.

[43] Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 1996.

[44] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 7 2009.

[45] Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *Int. Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *LNCS*, pages 474–486. Springer, 1992.

[46] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 46(3), Feb 2009.

[47] Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, 10 2007.

[48] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 379–390. Association for Computing Machinery, 2000.

[49] Zhe Chen, Yifan Wu, Ou Wei, and Bin Sheng. Deciding weak monitorability for runtime verification. In *Int. Conference on Software Engineering (ICSE'18)*, pages 163–164. ACM, 2018.

[50] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-based runtime verification with partial observability and resets. In *Int. Conference on Runtime Verification (RV'19)*, volume 11757 of *LNCS*, pages 165–184. Springer, 2019.

[51] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Int. Conference on Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.

[52] CLIPS. Website. http://clipsrules.sourceforge.net, 2017. Accessed: 2017-03-21.

[53] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, United States, 2009. IEEE Computer Society.

[54] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Formal Methods: Foundations and Applications*, pages 144–162. Springer International Publishing, 2018.

[55] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647–651. ACM, 2003.

[56] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[57] Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining*, SoftwareMining 2016, pages 3–10, New York, NY, United States, 2016. ACM.

[58] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of ω-languages. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 364–378, Berlin, Heidelberg, 2005. Springer.

[59] 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 *Proc. of the 12th Int. Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE Computer Society, 6 2005.

[60] ThirdEye Data. Website. https://thirdeyedata.io/, 2019. Accessed: Sep 2019.

[61] Datastream.io. Website. https://blog.ment.at/datastream-io-open-source-anomaly-detection-64db282735e0, 2019. Accessed: Sep 2019.

[62] Sandra De Amo, Arnaud Giacometti, and Waldecir Pereira Junior. Mining first-order temporal interval patterns with regular expression constraints. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 459–469, Berlin, Heidelberg, 2007. Springer, Springer.

[63] Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{\text{RV}}$—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.

[64] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, 18(2):205–225, 4 2016.

[65] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Advances in Database Technology - EDBT 2006*, pages 627–644. Springer Berlin Heidelberg, 2006.

[66] Volker Diekert and Paul Gastin. *First-order definable languages*, pages 261–306. Amsterdam University Press, 2008.

[67] Volker Diekert and Martin Leucker. Topology, monitorable properties and runtime verification. *Theoretical Computer Science*, 537:29–41, 2014.

[68] Volker Diekert, Anca Muscholl, and Igor Walukiewicz. A note on monitors and büchi automata. In *Int. Colloquium on Theoretical Aspects of Computing (ICTAC'15)*, volume 9399 of *LNCS*, pages 39–57. Springer, 2015.

[69] Laura K. Dillon, George Kutty, Louise E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Trans. Softw. Eng. Methodology*, 3(2):131–165, 4 1994.

[70] B. Ding, D. Lo, J. Han, and S. C. Khoo. Efficient mining of closed repetitive gapped subsequences from a sequence database. In *2009 IEEE 25th Intl. Conference on Data Engineering*, pages 1024–1035, 3 2009.

[71] Google Protocol Buffers Documentation. Website. https://developers.google.com/protocol-buffers/docs/overview#extensions, 2020. Accessed: Jun 2020.

[72] Steve Donoho. Early detection of insider trading in option markets. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 420–429, New York, NY, United States, 2004. ACM.

[73] Robert B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995.

[74] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[75] Drools. Website. http://www.jboss.org/drools, 2017. Accessed: 2017-03-21.

[76] Y. Du, J. Liu, F. Liu, and L. Chen. A real-time anomalies detection system based on streaming technology. In *2014 Sixth IHMSC*, volume 2, pages 275–279, Aug 2014.

[77] Murray Dunne, Giovani Gracioli, and Sebastian Fischmeister. A comparison of data streaming frameworks for anomaly detection in embedded systems. In *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, pages 30–33, 2018.

[78] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, United States, 1999. ACM.

[79] ECMA. The JSON data interchange syntax 2nd edition. ECMA 404, European Computer Manufacturers Association, Geneva, Switzerland, 12 2017.

[80] FY Edgeworth. Xli. on discordant observations. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 23(143):364–375, 1887.

[81] C. D. Edwards, D. J. Bell, R. E. Gladden, P. A. Ilott, T. C. Jedrey, M. D. Johnston, J. L. Maxwell, R. Mendoza, G. W. McSmith, C. L. Potts, B. C. Schratz, M. M. Shihabi, J. M. Srinivasan, P. Varghese, S. S. Sanders, and M. Denis. Relay support for the mars science laboratory mission. In *Conference on Aerospace*, pages 1–14. IEEE, 2013.

[82] Yahoo EGADS. Website. https://github.com/yahoo/egads, 2019. Accessed: Sep 2019.

[83] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[84] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification*, pages 40–59, Berlin, Heidelberg, 2009. Springer.

[85] Yliès 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, 6 2012.

[86] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, pages 141–175. IOS Press, 2013.

[87] Tom Fawcett and Foster Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 53–62, New York, NY, United States, 1999. ACM.

[88] H.G. Feichtinger. Discretization of convolutions and the generalized sampling principle. In *Progress in Approximation Theory*, pages 333–345, Boston; Toronto, 1991. Academic Press.

[89] Matthias Feurer and Frank Hutter. *Hyperparameter Optimization*, chapter 1, pages 3–33. Springer International Publishing, 2019.

[90] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.

[91] Alain Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3):129–135, Mar 1994.

[92] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[93] Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, August 2017.

[94] Gerard Frankowski, Marcin Jerzak, Maciej Miłostan, Tomasz Nowak, and Marek Pawłowski. Application of the complex event processing system for anomaly detection and network monitoring. *Computer Science*, 16(4):351–371, 2015.

[95] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 151–162, New York, NY, United States, 2011. ACM.

[96] GEM. Website. https://www.softwareag.com/gem/default.html, 2020. Accessed: Jun 2020.

[97] Ghosh and Reilly. Credit card fraud detection with a neural-network. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 3, pages 621–630, Jan 1994.

[98] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[99] Kalpana Gondi, Yogeshkumar Patel, and A. Prasad Sistla. Monitoring the full range of ω-regular properties of stochastic systems. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 105–119, Berlin, Heidelberg, 2009. Springer.

[100] Jean Goubault-Larrecq and Julien Olivain. A smell of ORCHIDS. In *Proc. of the 8th Int. Workshop on Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 1–20. Springer, 2008.

[101] N. Halbwachs, J. F. Héry, J. C. Laleuf, and X. Nicollin. Stability of discrete sampled systems. In *Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'20)*, volume 1926 of *LNCS*, pages 1–11. Springer, 2000.

[102] Sylvain Hallé. When RV meets CEP. In *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*, pages 68–91. Springer, 2016.

[103] Sylvain Hallé. *Event Stream Processing with BeepBeep 3: Log Crunching and Analysis Made Easy.* Presses de l'Université du Québec, 12 2018.

[104] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, 5 2000.

[105] Michael Reichhardt Hansen and Dang Van Hung. A theory of duration calculus with application. In *Domain modeling and the duration calculus*, volume 4710 of *LNCS*, pages 119–176. Springer, 2007.

[106] Abida Haque, Alexandra DeLucia, and Elisabeth Baseman. Markov chain modeling for anomaly detection in high performance computing system logs. In *Proceedings of the Fourth International Workshop on HPC User Support Tools*, HUST'17, pages 3:1–3:8, New York, United States, 2017. ACM.

[107] Klaus Havelund. Rule-based runtime verification revisited. *Int. J. Software Tools for Technology Transfer*, 17(2):143–170, 4 2015.

[108] Klaus Havelund and Rajeev Joshi. Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering: 16th International Conference on Formal Engineering Methods (ICFEM 2014)*, volume 8829 of *LNCS*, pages 187–202. Springer International Publishing, 2014.

[109] Klaus Havelund and Rajeev Joshi. Experience with rule-based analysis of spacecraft logs. In Cyrille Artho and Csaba Peter Ölveczky, editors, *Formal Techniques for Safety-Critical Systems: Third International Workshop (FTSCS 2014)*, volume 476 of *Communications in Computer and Information Science*, pages 1–16. Springer International Publishing, 4 2015.

[110] Klaus Havelund, Doron Peled, and Dogan Ulus. First order temporal logic monitoring with BDDs. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017)*, FMCAD '17, pages 116–123, Austin, TX, 2017. FMCAD Inc.

[111] L. Hofland and J. van der Linden. Software in mri scanners. *IEEE Software*, 27(4):87–89, July 2010.

[112] Dirk Holtwick. Website. https://pypi.org/project/xhtml2pdf/, 2020. Accessed: Jan 2020.

[113] G.J. Holzmann. Landing a spacecraft on mars. *IEEE Software*, 30(2):83–86, March 2013.

[114] John E Hopcroft and Richard M Karp. A linear algorithm for testing equivalence of finite automata. Technical report, Cornell University, 1971.

[115] Frank Höppner. Discovery of temporal patterns. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 192–203. Springer, 2001.

[116] Hogzilla ids Data. Website. http://ids-hogzilla.org/, 2020. Accessed: Jun 2020.

[117] O. Iegorov and S. Fischmeister. Mining task precedence graphs from real-time embedded system traces. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–260, April 2018.

[118] Oleg Iegorov, Reinier Torres, and Sebastian Fischmeister. Periodic task mining in embedded system traces. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 331–340, 4 2017.

[119] Information technology – Telecommunications and information exchange between systems – High-level data link control (HDLC) procedures. Standard, International Organization for Standardization, Geneva, CH, Jul 2002.

[120] Purush Iyer and Murali Narasimha. Probabilistic lossy channel systems. In *Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *LNCS*, pages 667–681. Springer, 1997.

[121] Jess. Website. http://www.jessrules.com/jess, 2017. Accessed: 2017-03-21.

[122] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. Runtime verification of LTL on lossy traces. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1379–1386, New York, NY, United States, 2017. ACM.

[123] JSON.org. Website. http://json.org/, 2018. Accessed: 2018-05-25.

[124] Po-shan Kam and Ada Wai-chee Fu. *Discovering Temporal Patterns for Interval-based Events*, pages 317–326. Springer, Berlin, Heidelberg, 2000.

[125] Sean Kauffman, Murray Dunne, Giovani Gracioli, Waleed Khan, Nirmal Benann, and Sebastian Fischmeister. Palisade: A framework for anomaly detection in embedded systems. *Journal of Systems Architecture*, 2020. In Press.

[126] Sean Kauffman, Murray Dunne, Giovani Gracioli, Waleed Khan, Nirmal Benann, and Sebastian Fischmeister. Palisade: A framework for anomaly detection in embedded systems dataset, 2020.

[127] Sean Kauffman and Sebastian Fischmeister. Mining temporal intervals from real-time system traces. In *International Workshop on Software Mining (SoftwareMining'17)*, pages 1–8. IEEE, 2017.

[128] Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. What can we monitor over unreliable channels? *International Journal on Software Tools for Technology Transfer*, 2020. Accepted.

[129] Sean Kauffman, Klaus Havelund, and Rajeev Joshi. nfer–a notation and system for inferring event stream abstractions. In *International Conference on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 235–250. Springer, 2016.

[130] Sean Kauffman, Klaus Havelund, Rajeev Joshi, and Sebastian Fischmeister. Inferring event stream abstractions. *Formal Methods in System Design*, 53:54–82, 2018.

[131] Sean Kauffman, Rajeev Joshi, and Klaus Havelund. Towards a logic for inferring properties of event streams. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA'16)*, volume 9953 of *LNCS*, pages 394–399. Springer, 2016.

[132] Sean Kauffman, Carlos Moreno, and Sebastian Fischmeister. Static transformation of power consumption for software attestation. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'16)*, pages 188–194. IEEE, 2016.

[133] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering*, pages 81–90. IEEE Computer Society Press, 1994.

[134] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000.

[135] Steven M. Kearns. Extending regular expressions with context operators and parse extraction. *Software: Practice and Experience*, 21(8):787–804, 1991.

[136] Eamonn Keogh, Jessica Lin, Sang-Hee Lee, and Helga Van Herle. Finding the most unusual time series subsequence: algorithms and applications. *Knowledge and Information Systems*, 11(1):1–27, 1 2007.

[137] Mark Kramer. Nonlinear principal component analysis using autoassociative neural networks. *American Institute of Chemical Engineers*, 37(2):233–243, 1991.

[138] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proc. of the 6th Int. Workshop on Networking Meets Databases (NetDB'11)*, pages 1–7. ACM, 2011.

[139] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms.* Wiley-Interscience, New York, NY, USA, 2004.

[140] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 11 2001.

[141] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(3):408–429, 07 2001.

[142] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, volume 83 of *Information Processing*, pages 657–668. Elsevier, 1983.

[143] Phillip A Laplante et al. *Real-time systems design and analysis.* Wiley New York, 2004.

[144] Tien-Duy B Le and David Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 331–340. IEEE, 2015.

[145] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306, Berlin, Heidelberg, 2009. Springer, Springer.

[146] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 122–135, Berlin, Heidelberg, 2010. Springer.

[147] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92, Nov 2015.

[148] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. In *Int. Conference on Runtime Verification (RV'19)*, volume 11757 of *LNCS*, pages 273–291. Springer, 2019.

[149] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[150] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *Int. Conference on Distributed Computing Systems Workshops (ICDCSW'07)*, IEEE, pages 67–67, 2007.

[151] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 2–11, New York, NY, United States, 2003. ACM.

[152] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, 10 1990.

[153] Xue Liu, Hui Ding, Kihwal Lee, Lui Sha, and Marco Caccamo. Feedback fault tolerance of real-time embedded systems: Issues and possible solutions. *SIGBED Rev.*, 3(2):23–28, 4 2006.

[154] Alessio Lomuscio, Wojciech Penczek, and Hongyang Qu. Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. In *Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 659–666. ACM, 2010.

[155] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte. A performance comparison of open-source stream processing platforms. In *2016 IEEE GLOBECOM*, pages 1–6, Dec 2016.

[156] Étienne Lozes and Jules Villard. Reliable contracts for unreliable half-duplex communications. In Marco Carbone and Jean-Marc Petit, editors, *Web Services and Formal Methods*, pages 2–16, Berlin, Heidelberg, 2012. Springer.

[157] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[158] Eduard Marin, Dave Singelée, Bohan Yang, Ingrid Verbauwhede, and Bart Preneel. On the feasibility of cryptography for a wireless insulin pump system. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 113–120, New York, NY, USA, 2016. ACM.

[159] F. Marvasti, M. Analoui, and M. Gamshadzahi. Recovery of signals from nonuniform samples using iterative methods. *IEEE Transactions on Signal Processing*, 39(4):872–878, Apr 1991.

[160] Florent Masseglia, Maguelonne Teisseire, and Pascal Poncelet. Sequential pattern mining. In *Encyclopedia of Data Warehousing and Mining*, pages 1028–1032. IGI Global, 2005.

[161] C. McCarthy, K. Harnett, and A. Carter. Characterization of potential security threats in modern automobiles: A composite modeling approach. Technical report, National Highway Traffic Safety Administration, Washington, 2014.

[162] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[163] Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, 6 2012.

[164] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *Blackhat USA*, pages 1–91. IOActive, 2015.

[165] Mitre. Common attack pattern enumeration and classification. https://capec.mitre.org/data/definitions/1000.html, 2018. Accessed: Sep 2018.

[166] Y. Mo and B. Sinopoli. Secure control against replay attacks. In *2009 47th Annual Allerton Conference on Communication, Control, and Computing*, pages 911–918, Sept 2009.

[167] Fabian Mörchen. Unsupervised pattern mining from symbolic temporal data. *ACM SIGKDD Explorations Newsletter*, 9(1):41–55, 6 2007.

[168] Carlos Moreno, Sebastian Fischmeister, and M. Anwar Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis.

In *Proc. of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 77–88, New York, USA, 2013. ACM.

[169] Carlos Moreno, Sean Kauffman, and Sebastian Fischmeister. Efficient program tracing and monitoring through power consumption – with a little help from the compiler. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*, pages 1556–1561. IEEE, 2016.

[170] Ben C. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18:10–19, 1985.

[171] Subhojeet Mukherjee, Hossein Shirazi, Indrakshi Ray, Jeremy Daily, and Rose Gamble. Practical DoS attacks on embedded networks in commercial vehicles. In *Information Systems Security*, pages 23–42. Springer, 2016.

[172] Apurva Narayan, Greta Cutulenco, Yogi Joshi, and Sebastian Fischmeister. Mining timed regular specifications from system traces. *ACM Trans. Embed. Comput. Syst.*, 17(2):46:1–46:21, 1 2018.

[173] Apurva Narayan, Sean Kauffman, Jack Morgan, Guy Martin Tchamgoue, Yogi Joshi, Chris Hobbs, and Sebastian Fischmeister. System call logs with natural random faults: Experimental design and application. In *International Workshop on Silicon Errors in Logic – System Effects (SELSE'17)*, SELSE-13. IEEE, 2017.

[174] NATS. Website. https://nats.io/, 2018. Accessed: Jan 2018.

[175] 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, United States, 2013. ACM.

[176] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[177] Jian Pei, Jiawei Han, B. Mortazavi-Asl, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings 17th International Conference on Data Engineering*, pages 215–224, 4 2001.

[178] Doron Peled and Klaus Havelund. Refining the safety–liveness classification of temporal properties according to monitorability. In *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *LNCS*, pages 218–234. Springer, 2019.

[179] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

[180] Wuxu Peng and Kia Makki. Lossy communicating finite state machines. *Telecommunication Systems*, 25(3):433–448, Mar 2004.

[181] Jonathan Petit, Bas Stottelaar, and Michael Feiri. Remote attacks on automated vehicles sensors : Experiments on camera and lidar. In *Black Hat Europe*, pages 1–13, 2015.

[182] Tomáš Pevný. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102(2):275–304, Feb 2016.

[183] James Pickands. Statistical inference using extreme order statistics. *The Annals of Statistics*, 3(1):119–131, 1975.

[184] Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014.

[185] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 573–586, Berlin, Heidelberg, 2006. Springer.

[186] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71. IEEE Computer Society, 2007.

[187] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *Int. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 270–285. ACM, 2010.

[188] QNX. *QNX Operating System: system architecture*. QNX Software Systems Ltd., 1997.

[189] RabbitMQ. Website. http://www.rabbitmq.com/, 2018. Accessed: Jan 2018.

[190] Redis. Website. https://redis.io/, 2018. Accessed: May 2018.

[191] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *28th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*, pages 658–663, Nov 2013.

[192] Giles Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.

[193] Giles Reger, Howard Barringer, and David Rydeheard. Automata-based pattern mining from imperfect traces. *SIGSOFT Softw. Eng. Notes*, 40(1):1–8, 2 2015.

[194] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, pages 596–610, Berlin, Heidelberg, 2015. Springer.

[195] Kelly Rollick, Allan Roczko, and Leslie Mitchell. Combustible gas detector sensor drift: Catalytic vs. infrared. *InTech Magazine*, Aug 2010.

[196] Grigore Rosu and Saddek Bensalem. Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In *18th Int. Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 263–277. Springer, 2006.

[197] M. Roudjane, D. Rebaïne, R. Khoury, and S. Hallé. Real-time data mining for event streams. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 123–134, 2018.

[198] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[199] S. Safra. On the complexity of ω-automata. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 319–327, 10 1988.

[200] K. Sauer and J. Allebach. Iterative reconstruction of bandlimited images from nonuniformly spaced samples. *IEEE Transactions on Circuits and Systems*, 34(12):1497–1506, 12 1987.

[201] M. Seiter, H. J. Mathony, and P. Knoll. Parking assist. In *Handbook of Intelligent Vehicles*, pages 829–864. Springer, 2012.

[202] W. Shi, Y. Zhu, T. Huang, G. Sheng, Y. Lian, G. Wang, and Y. Chen. An integrated data preprocessing framework based on Apache Spark for fault diagnosis of power grid equipment. *Journal of Signal Processing Systems*, 86(2):221–236, Mar 2017.

[203] Donghyun Shin. A platform for generating anomalous traces under cooperative driving scenarios. Master's thesis, University of Waterloo, 2018.

[204] Siddhi. Website. `https://siddhi.io/`, 2019. Accessed: Jun 2020.

[205] Radu Siminiceanu, Ricky W. Butler, and César A. Muñoz. Experimental evaluation of a planning language suitable for formal verification. In *5th Int. Workshop on Model Checking and Artificial Intelligence (MoChArt'08)*, volume 5348 of *LNCS*, pages 132–146, Berlin, Heidelberg, 2009. Springer.

[206] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[207] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, Sep 1994.

[208] A. Prasad Sistla, Miloš Žefran, and Yao Feng. Monitorability of stochastic dynamical systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 720–736, Berlin, Heidelberg, 2011. Springer.

[209] Damon Snyder. On-line intrusion detection using sequences of system calls. Master's thesis, Florida State University, 2001.

[210] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, J. Ingram, and Sadi E. Seker. Online anomaly detection for multi-source VMware using a distributed streaming framework. *Softw. Pract. Exper.*, 46(11):1479–1497, November 2016.

[211] F. Song, Y. Diao, J. Read, A. Stiegler, and A. Bifet. EXAD: A system for explainable anomaly detection on big data traces. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 1435–1440, 2018.

[212] Henry Stark. *Image recovery: theory and application*. Elsevier, 1987.

[213] Brandon Stephens. I/o latency in the linux storage stack. Master's thesis, Florida State University, 2017.

[214] Scott D Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A Smolka, and Erez Zadok. Runtime verification with state estimation. *Runtime Verification*, 11:193–207, 2011.

[215] Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Gray-box monitoring of hyperproperties. In *Formal Methods (FM'19)*, volume 11800 of *LNCS*, pages 406–424. Springer, 2019.

[216] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proc. of the 32Nd VLDB*, pages 187–198, 2006.

[217] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50. ACM, 2011.

[218] B. P. Swenson and G. F. Riley. A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 44–52, July 2012.

[219] A. Taylor, S. Leblanc, and N. Japkowicz. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 130–139, Oct 2016.

[220] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[221] Esper Tech. Website. http://www.espertech.com/, 2019. Accessed: Jun 2020.

[222] Hetal Thakkar, Barzan Mozafari, and Carlo Zaniolo. Designing an inductive data stream management system: the stream mill experience. In *Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 79–88. ACM, 2008.

[223] Michiel van Genuchten and Les Hatton. Compound annual growth rate for software. *IEEE Software*, 29(4):19–21, July 2012.

[224] Guido van Rossum. *Python Language Reference*. Python Software Foundation, 2020.

[225] J. Wang and J. Han. BIDE: efficient mining of frequent closed sequences. In *Proceedings. 20th International Conference on Data Engineering*, pages 79–90, March 2004.

[226] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Recent Advances in Intrusion Detection*, pages 203–222, Berlin, Heidelberg, 2004. Springer.

[227] Z. Wang, M. H. Zaki, and S. Tahar. Statistical runtime verification of analog and mixed signal designs. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, pages 1–6, 11 2009.

[228] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, May 1999.

[229] Marc Weber, Georg Wolf, Eric Sax, and Bastian Zimmer. Online detection of anomalies in vehicle signals using replicator neural networks. In *6th Embedded Security in Cars USA (escar USA)*, pages 1–14, Jun 2018.

[230] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Symposium on Principles of Programming Languages (POPL'86)*, pages 184–193. ACM, 1986.

[231] Weng-Keen Wong, Andrew W Moore, Gregory F Cooper, and Michael M Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 808–815, 2003.

[232] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, United States, 2006. ACM.

[233] Xilinx. Zynq-7000 all programmable SoC ZC706 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html, 5 2017. accessed: 2017-05-12.

[234] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 166–177. SIAM, 2003.

[235] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.

[236] Mohammad Mehdi Zeinali Zadeh, Mahmoud Salem, Neeraj Kumar, Greta Cutulenco, and Sebastian Fischmeister. SiPTA: Signal processing for trace-based anomaly detection. In *Proc. of the International Conference on Embedded Software (EMSOFT)*, EMSOFT '14, pages 6:1–6:10, New York, NY, United States, Oct. 2014. ACM.

[237] Mohammed J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1):31–60, 2001.

[238] Haopeng Zhang, Yanlei Diao, and Neil Immerman. Recognizing patterns in streams with imprecise timestamps. *Proc. VLDB Endow.*, 3(1–2):244–255, September 2010.

[239] Yuxun Zhou, Han Zou, Reza Arghandeh, Weixi Gu, and Costas J Spanos. Non-parametric outliers detection in multiple time series a case study: Power grid data analysis. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 4605–4612, 2018.