CAMPUS
DE EXCELENCIA
INTERNACIONAL

UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S.I. INFORMÁTICOS

Master in Software and Systems

MASTER THESIS

Towards a stream-based monitoring language for asynchronous systems

Author: Felipe Gorostiaga

Supervisor: Lars-Åke Fredlund

Co-Supervisor: César Sánchez

MADRID, JULY, 2018

Abstract

In this thesis, we study the problem of monitoring rich properties of real-time event streams, and propose a solution based on Stream Runtime Verification.

Stream Runtime Verification (SRV) is a specification formalism where observations are described as streams of data computed from input streams of data, which allows us to cleanly separate the temporal dependencies between events and the concrete operations that are performed during the monitoring. However, Stream Runtime Verification languages typically assume that all streams share a global synchronous clock and input events arrive in a synchronous manner.

In this thesis we generalize the time assumption to cover streams whose events are stamped from a real-time domain, but keep the essential explicit time dependencies present in previous synchronous SRV languages. The resulting formalism, which we call *Striver*, shares with synchronous SRV the simplicity of the separation between the timing reasoning and the data domain.

We demonstrate how *Striver* can serve as a general language to express other real-time monitoring languages by showing translations from other logics and specification languages for (piece-wise constant) signals and timed event streams. Finally, we report an empirical evaluation of an implementation of *Striver* and present a real case of a system for the monitorization of cloud applications where this implementation has been sucessfully used.

Resumen

En esta tesis, estudiamos el problema de monitorizar propiedades sobre flujos de eventos en tiempo real, y proponemos una solución basada en *Stream Runtime Verification*.

Stream Runtime Verification (SRV) es un formalismo de especificación en donde las observaciones se describen como flujos de datos computados a partir de flujos de datos de entrada, lo que permite establecer una separación limpia entre las dependencias temporales de los eventos y las operaciones concretas que se llevan a cabo durante la monitorización. Sin embargo, los lenguajes de *Stream Runtime Verification* típicamente suponen que todos los flujos de entrada comparten un reloj global sincronizado, y que los eventos se reciben de manera síncrona.

En esta tesis generalizamos las suposiciones sobre el tiempo para considerar flujos de eventos en donde cada evento viene acompañado por el instante de tiempo real en el cual se produjo, manteniendo explícitas las dependencias temporales esenciales características de lenguajes de SRV preexistentes. El formalismo resultante, al cual llamamos *Striver*, comparte con los lenguajes de SRV síncronos la simplicidad de la distinción entre los razonamientos temporales y el dominio de datos

Mostramos que *Striver* puede utilizarse como un lenguaje general para expresar otros lenguajes de monitorización de tiempo real, presentando traducciones de otras lógicas y lenguajes de especificación para señales (definidas por partes) y flujos de eventos temporales. Finalmente, reportamos una evaluación empírica de una implementación de *Striver* y presentamos un caso real de un sistema para la monitorización de aplicaciones en la nube en donde dicha implementación ha sido utilizada con éxito.

Contents

Chapter 1

Introduction

1.1 Microservice architecture and cloud systems

In a private meeting at the Compaq Computer offices in 1996, a group of techonology executives made the first recorded use of the term “cloud computing” to describe their perceived direction of the internet business: many communication and collaboration applications were migrating into the so-called “internet cloud” [1] posing new challenges and presenting new business oportunities [2].

A decade later, the term was popularized by big companies such as Google and Amazon who employed it to describe the new paradigm in which people were increasingly accessing software and files over the Web instead of on their desktops [3], but a more precise definition of it was only released five years later, in September 2011, by the National Institute of Standards and Technology (NIST) [4], stating that:

Definition Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

According to the NIST, the consumer of a cloud-computing system should be able to require and be provisioned of computing capabilities, without the need of human interaction with the service providers (“on-demand self-service”), the system’s capabilities should be accessible over the network through standard mechanisms (“broad netork access”), the service provider should manage a pool of resources serving multiple consumers and enabling the use of load-balancers and transparent dynamic resource assignment (“resource pooling”), it should be easy for the service provider to provision its capabilities elastically, scaling up or down automatically according to the demand and thus giving the sensation of unlimited resources for the user (“rapid elasticity”) and lastly, for each service there should exist a metering capability to optimize the resources (“measured service”).

These requirements of an emerging technology which was rapidly gaining momentum increased

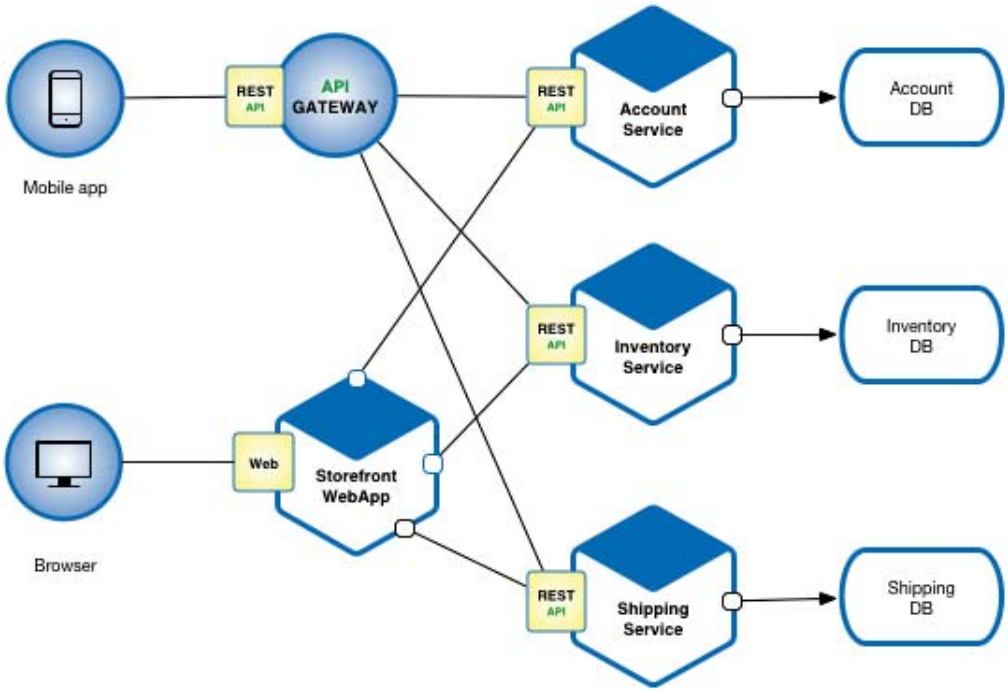


Figure 1.1: Example of a microservices architecture for an e-commerce application [5]

the adoption of Hardware Virtualization for servers in the cloud, which would allow running multiple computing stacks on a single physical host (Virtual Machines). Later, the trend became to virtualize applications, where an "Application Container" ran on top of a guest operating system.

This model leveraged a Microservices architecture, in which an entreprise-scale system is assembled from a distributed set of small, stateless, self-contained, loosely coupled services that communicate with each other using a communication protocol and a set of APIs, allowing services to be reused and combined to reflect changing business priorities, as is depicted in figure 1.1. The NIST provides a precise definition of what these microservices are [6]:

Definition A microservice is a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology.

Developing a tool to be run on the cloud makes it possible to leave its deployment in the hands of third-party cloud infrastructure providers, which lets organizations focus on their core businesses instead of spending resources on computer infrastructure and maintenance [7].

The microservices that make up a cloud system can be developed in different languages and some of them are third parties' tools which are maintained by external software vendors, and whose source code is not under control of the company using it. This is why the integration of the microservices is performed using standard communication protocols such as SOAP [8], WSDL [9] or the exposure of REST APIs using, for example, OpenAPI [10].

At the same time, it is hard for the applicaiton admins to get feedback on their behavior, so the standard way to get insight and detect bugs in the cloud system is by inspecting the microservices'

logfiles.

The intrinsic concurrency and heterogeneity of components in a microservices architecture makes it nearly impossible to prove static properties of the system as a whole, rendering it a highly suitable scenario to be monitored online using Runtime Verification techniques, as those described in Section 1.2.

1.2 Runtime Verification (RV)

Software plays a central role in everyday life. From the applications in our smartphones to the systems controlling trains and traffic lights; we live in constant touch with programs that control and are deeply affected by their environment. Most of the systems which interact directly with humans are reactive systems [11], and since they are affected by the environment in runtime, these systems often cannot be abstracted to a sequential model of computation, and thus have to be treated as concurrent systems.

An error that goes undetected could lead to the malfunctioning, downtime or faulty behavior of the system, which in some cases translates into millons of dollars, or even human lives losses. Such is the case with the infamous bug found on a computer-controlled radiation therapy machine, the Therac-25, which massively overdosed six people, causing their death or serious injury [12], illustrating that it is essential to check that critical or potentially hazardous software works in a correct, secure, and reliable way.

To check if a program behaves as expected, the developer would typically run it in a controlled environment with a battery of inputs and check that the corresponding obtained outputs satisfy the specification, trying to comprise as many types of scenarios as possible. This approach is straightforward and easy to craft even for unexperienced developers. However, it is usually impossible or unfeasible to exercise and test every possible scenario. Furthermore, correctness of systems may often involve not only input and output behavior but also the execution of the system itself. Also, in the words of Edsger Dijkstra, testing shows the presence, but not the absence of bugs.

The traditional approach to prove a system correct is to draw upon logics and formal reasoning and apply these techniques to statically analyse the behavior of the system. To do so, the engineers model the system using a finite-state representation, express properties as formulas on a logic of their choice, and then proceed to either prove that these formulas hold for the system under analysis; or provide a counter-example showing a witness trace for which the property does not hold.

Some of the most widely used logics for the verification of reactive systems are *temporal logics*, whose application to complex computational systems was proposed by Pnueli in the late '70s [13], including Linear Temporal Logic (LTL) [14] and Computation Tree Logic (CTL) [11]. In LTL, each moment in time has a single successor, which allows specifying relations in time and thus easily express finite control properties. This logic will be described in more detail in Chapter 2. On the other hand, CTL has a branching, tree-like structure underneath, where time may split into

alternative courses to express the possible ways the program can potentially proceed.

Static verification of a program proves that all its possible runs satisfy a specified property. If a system is large, the amount of cases that need to be taken into account to prove the system correct can be too high to handle, which renders this technique a complex, heavy task that is hard to maintain, and ultimately unsuitable for large systems.

The necessity of highly specialized logicians to craft formulas and proofs adds up to the cost of the system, and grows very rapidly in terms of the size and complexity of the system under analysis. Also, a high degree of manual effort in the modeling and proof of properties has to be redone every time the code is updated, making it inappropiate for evolving programs. Additionally, a model of the program is verified instead of the running code, and this model can diverge from the behavior of the actual system.

In some sense lying between testing and static verification, we find the field of *Runtime Verification* (RV). Martin Leucker and Christian Schallhart give a formal definition of Runtime Verification [15]:

Definition Runtime Verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.

Runtime Verification is esentially a lightweight formal method that studies the problem of whether a single trace from the system under analysis satisfies a formal specification.

From the point of view of coverage, model-checking and static forms of verification must consider all possible executions of the systems while RV only considers the traces observed. In this manner, RV sacrifices completeness but offers a readily applicable formal method that can be combined with testing or debugging. Surveys on Runtime Verification can be found in [16, 17], and also in the recent book [18].

Runtime Verification is performed dynamically besides the analysed system, allowing the verifier to act whenever the system behaves incorrectly and correct its misconduct or stop its execution completely.

As opposed to the static analysis of a program, the runtime verifier (hereinafter “the monitor”) and the system under verification may evolve at different paces, and updating one of them does not necessarily require to change its counterpart. Even more, since Runtime Verification deals only with observed executions as they are generated by the real system, it can be applied to black-box analysis, for which no model or even source code is available, as long as it is appropiately instrumented.

Runtime Verification techniques deal with the executing code, eliminating the gap between a program and the model to be verified, an inherent issue of the static verification approach. Furthermore, the static analysis of a program requires considering all the possible states the underlying system can go through (the so-called “state explosion” problem), while a monitor only needs to take into account those states that the system effectively reaches, usually avoiding any memory problems if only a small part of the execution history needs to be stored.

Monitors will most likely run in specialized hardware such as FPGAs or will share the running system's computational resources and should influence the system as little as possible. Also, monitors will usually run for long periods of time and have to deal with large amounts of input data. This is why the complexity and efficiency of the monitors is of great interest, being of extreme importance to be able to calculate the amount of memory required by a monitor beforehand in static time. Also, it is often necessary to instrumentalize the system under scrutiny to retrieve information of its execution seeking to interfer with it as little as possible.

The monitor of a well-deployed Runtime Verification framework should be able to point bugs in the system under scrutiny, but it will only do so once the faulty behavior has been performed.

An important question that remains when a monitoring approach to verification is used, is what happens when a problem is discovered by the checker. The effectiveness of recovery actions often depends on the assumption that system execution does not progress beyond the point of violation, which impplies that checking should be synchronous and the system is effectively stopped while checking is performed, an assumption that may have a serious impact on the response time of the system.

Related to this issue is the fact that instrumentation and shared computational resources influence the response time of the system, which may be of special concern in critical real-time software.

1.2.1 Related work

Early specification languages proposed in RV were based on boolean temporal logics [19–21], metric temporal logics [22], regular expressions [23], timed regular expressions [24], rules [25], data automata [26], quantified event automata [27], signal temporal logics [28], rule-based logics [29], or rewriting [30]. Temporal testers [31] were later proposed as a monitoring technique for LTL based on Boolean streams.

The technique of defining specifications by declaring the dependencies between output streams (results) and input streams (observations) is called *Stream Runtime Verification* (SRV), and was first proposed in *LOLA* [32].

The main idea of SRV is that the same sequence of operations performed during the monitoring of a temporal logic formula can be followed to compute statistics of the input trace, if the data type and the operations are changed. SRV was initially conceived for monitoring synchronous systems, where all observations proceed in cycles.

Specifications in *LOLA* resemble a programming language, making it more suitable for typical engineers and helping improve the learning curve for programmers, as opposed to previous RV languages, where it is necessary to have a logic mindset and craft complicated logic formulas to describe a property. *LOLA* is described in more depth in Chapter 2.

Copilot, a real-time stream-based runtime monitor that generates small constant-time and constant-space C programs [33] is an example of an industrial implementation of a stream Runtime Verification tool based on *LOLA*.

The work [34] presents an asynchronous evaluation engine for a simple event stream language for timed events, based on a collection of building blocks that compute aggregations. This language does not allow explicit time references and offsets. Moreover, recursion is not permitted and all recursive computations are encapsulated implicitly in the building blocks.

A successor work of [34] is *TeSSLa* [35] which allows recursion and offers a smaller collection of building blocks.

Another similar work is RTLola [36], which also aims to extend SRV from the synchronous domain to timed streams.

These formalisms fail at solving the problem of monitoring a system running on the cloud for different reasons.

In [14], the time is somewhat irrelevant and the focus is set on the order of the events generated by the system. This makes it hard –if not impossible– to compute statistics and use them on the specification. This same issue affects the proposal of [26] and [27], where the relative order of the events is the core concept, allowing to express the correctness of a trace of events, but the time at which these were generated is ignored and not accessible by the specifier.

Signal Temporal Logics focus on the definition of properties based on signals in a dense time domain, but it is not possible to consider individual events.

On the other hand, *LOLA* and Copilot differ from our approach since they expect all the sources to feed exactly one event per tick, an assumption which doesn't hold in general on distributed (in particular, cloud) systems.

The rule-based tool HAWK [29] is stream-based but is not expressive enough to define derived metrics from input events and predicate upon them. It also resembles logic programming, which might seem odd to the usual mindset of engineers.

TeSSLa precludes explicit offset dependencies, and the target application domain is hardware based monitoring.

In RTLola defined streams are computed at predefined periodic instants of time, collecting aggregations between these predefined instants using building blocks. In this manner, the output streams in RTLola are *isochronous*, a term borrowed from telecomunications and signal processing where an isochronous signal is one in which events happen at regular intervals.

1.3 Complex Event Processing (CEP)

In an event-driven information system, some events are said to occur as a result of witnessing a specific sequence of simpler events. As an example, consider the TCP three-way handshake. In order for a TCP connection to be considered successfully established, the client and the server must exchange packets in a three steps protocol, depicted in figure 1.2:

- The client sends a SYN packet to the server

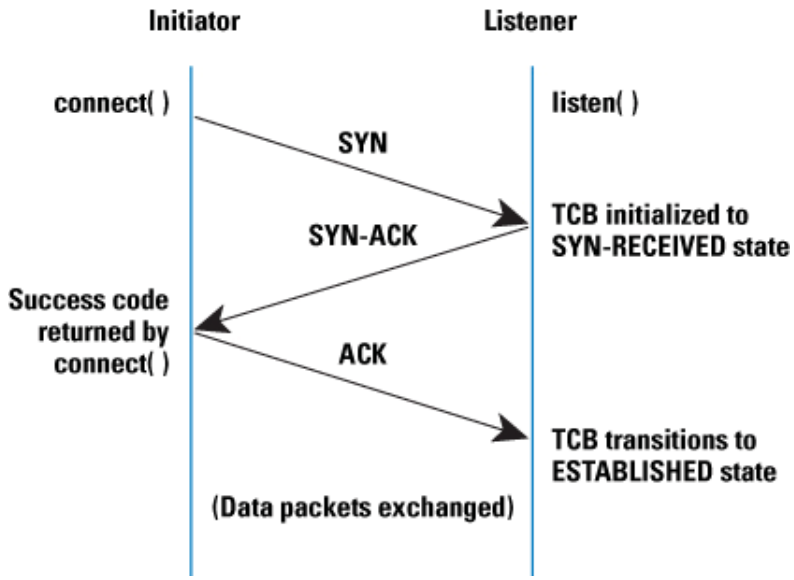


Figure 1.2: The packet exchanges in a successful TCP handshake protocol [37]

- The server replies with a SYN/ACK packet
- The client receives the SYN/ACK from the server and replies with an ACK packet

If the packets sent through the network are considered atomic events and this sequence of messages is observed between two nodes, then the capabilities of a Complex Event Processing system allow the inference of a new composite event stating that a connection between these nodes has been sucessfully established.

This composite event can in turn be part of a higher-level event, for example the beginning of an HTTP session, allowing the system admin to reason at a more complex level without the need of working with atomic events, enhancing the expresiveness of predicates, making them easier to express and consequently helping to achieve a less error-prone and more robust and reliable application.

Complex events carry the information of which events are the proof of its occurrence, recursively referring to a set of simpler events, and ultimately reaching atomic events as the base evidence.

1.3.1 Related work

The field of Complex Event Processing was born to enhance the expressivity of standard Runtime Verification event-driven tools, thus providing a way to abstract complex specifications from low level events, and let the user focus on higher-level domain specific properties.

A general-purpose declarative language to manipulate event traces based on the composition of processors can be found in [38]. Following the same idea, BeepBeep3 [39] provides a tool to perform queries over event traces using composing event processors.

In [40], the authors present a complex event language designed to generate high level events based on RFID readings, which can later be fed to external monitoring applications.

In [41], the authors show the use of an event stream processor to progressively infer high level events suitable for activity recognition, based on sensors of electrical consumption. The authors of [42] also present a language to infer complex events from RFID readings in enterprise information systems, improving the discovery of more actionable information.

Oracle CEP [43] is an open architecture to filter, correlate and process and publish events on runtime, which is based on a continuous query langauge (Oracle CQL) and provides a visual development environment as well as standard Java-based tooling.

All these tools have been designed with the sole purpose of inferring complex events, but none of them provides a logic to reason about the observed system.

Such logic is expected to be implemented using other tools, forcing the admins to master and integrate different many langauges to express properties. Plus, most of the aforementioned Complex Event Processing tools do not provide languages capable of expressing metric computations and work with them, even though they might be part of the verification tool following the workflow, and a complex event could be the result of certain value going beyond a threshold.

1.4 Time-Series DataBases (TSDB)

A time series is a sequence collected regularly in time, where each entry represents a value at a point of time. This kind of data is patricularly useful for understanding the underlying forces and structure that produced it, extract a model which fits such data, and afterwards proceed to the inference of information by mining it.

A metric is the measurement of a quantitative property of a system, which changes over time. For example, the CPU load and the inner temperature are common metrics of a machine.

Usually, the values of metrics are retrieved periodically at a high rate, making TSDB the perfect choice to store their values. It is often useful to aggregate and correlate the measured values over time, under the assumption that the sampling points make up a discretization of continuous signals, whose values remain pretty much constant inbetween. Therefore, TSDBs offer capabilities to get statistics out of sampled signals, and derive new signals from them.

Most of the industry standard Time Series Databases are shipped along with visualization tools, which allow the user to define how to scrape the values of a metric from incoming samples, and then plot the generated signal to a graphic ranging over time. Industrial Time Series systems usually come with a monitor which can be configured to periodically check and raise an alarm whenever a signal behaves in a certain way, for example, when its value decreases to under a certain threshold, or when it is stalled for a certain amount of time.

The industrial approach to check a property online using TSDB is to synthesize a query from the specification, and on runtime, persist the incoming events into the database and periodically check that the specification holds for the witnessed data. This setup is depicted in Figure 1.3.

Using this technique, queries can be modified on runtime; but this advantage also makes it nec-

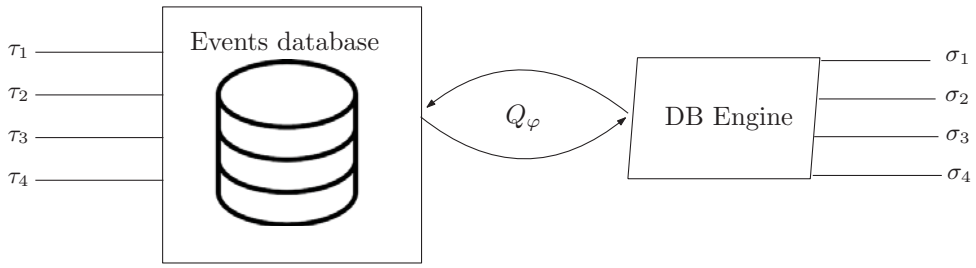


Figure 1.3: Online checking of a property using TSDB

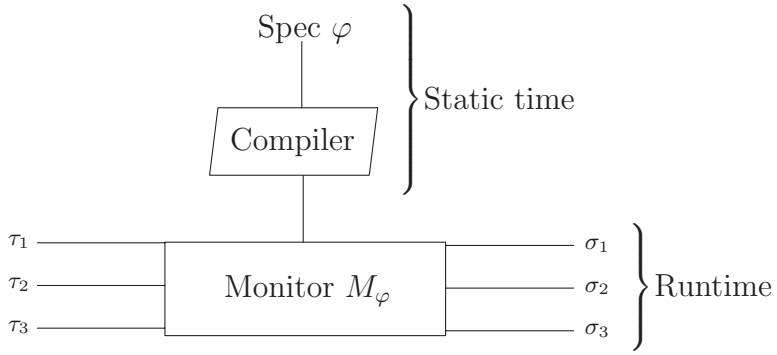


Figure 1.4: Online checking of a property using Runtime Verification

essary to store all the data, increasing the database size over time, subsequently requiring more resources as the system under scrutiny runs and resulting in a detriment on the database engine performance, penalizing both the high input events ratio and the duration of the monitorization.

Time Series Databases come in handy to monitor the behavior of a running system, in a way which is natural on the ad-hoc specification language of the industrial platforms, but is cumbersome or impossible to express in typical RV tools. On the other hand, since TSDBs are oriented to work with periodically sampled signals, an incoming event is usually not given an entity by itself, making it hard or impossible to express typical RV conditions.

Time series databases are often used in the fields of big data and data mining, so they are expected to store a large amount of data. This is why it is crucial for a TSDB to be efficient at storing and retrieving values over time.

However, the set up for Runtime Verification differs slightly from those of big data and data mining, because queries are known beforehand and they are executed on runtime with the data points being continually processed.

Using Stream Runtime Verification techniques, we syntheshize a monitor given a specification, which processes the input data and computes the results in real time. This setup is depicted in Figure 1.4

Because of this, a Runtime Verification tool can be developed to work with no permanent storage, getting rid of processed and uninteresting events, and with a memory consumption bounded by the queries that make up the specification of the monitor. As a consequence, monitors can run for long periods of time and handle a high amount of events without a loss in performance. However, the specification is fixed in static time and cannot be changed in runtime.

Nevertheless, even though storing the events generated by the system in a permanent database and periodically executing queries over it in order to check properties is inefficient, having this information available for an offline analysis may be of use once the monitor has pointed out which events are the ones which support the fact that a specified predicate was violated.

1.4.1 Related work

Since time series are present in many activities and in particular in business activities, industrial time series databases are widely used, and are usually part of a mature system comprising a stack of tools which do a specific task and are designed to complement each other, in microservices fashion architectures.

Such is the case of the TICK stack [44], developed and maintaned by InfluxData [45], with InfluxDB [46] as its time series database. The same design is followed by the Elastic stack (formerly, the ELK stack) [47], developed by Elastic [48] and with Elasticsearch [49] as the time series database.

Most of the academia works in time series databases are oriented to efficently handle large amounts of data, speeding up the execution of queries offline.

In [50], the authors propose a method to handle high dimensionality data by dimensionality reduction, effectively agilizing the execution of queries over large time series databases. In [51], it is shown how to locate patterns using a probabilistic approach over time series databases.

1.5 RV+CEP+TSDB for the monitorization of cloud systems

Performing Runtime Verification in the context of a cloud application developed following a microservices architecture just like those described in section 1.1 falls in the intersection of Runtime Verification, Complex Event Processing and Time Series Databases, and consequently poses the need of borrowing concepts from these three big areas of computer science, and putting them together in a harmonic and consistent way.

The monitor of a cloud system has to be ready to receive events from many sources at a different pace, in a scenario inherently asynchronous; with events arriving at arbitrary points in increasing time but with a global clock shared by all the microservices.

The cloud system under scrutiny is instrumented in the least intrusive way possible to send data about its execution to the monitor; which is deployed as an independent microservice, as depicted in Figure 1.5.

In order to gather information about the state of the machines running the different services, it is common to run a daemon on each of them which would report its metrics periodically. The gathered data can be naturally thought of and processed as time series, and can be correlated with the events containing application logic information sent by the instrumentalized microservices.

Our goal was to develop a Runtime Verification tool specifically designed to monitor systems in

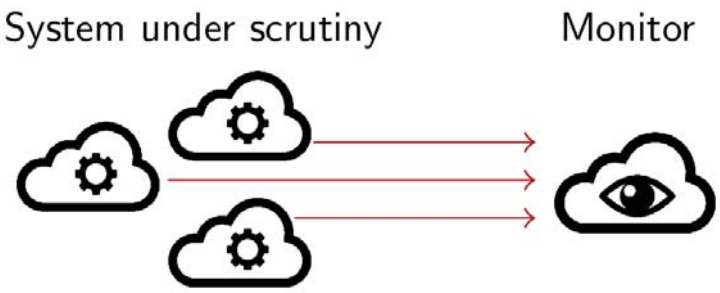


Figure 1.5: Monitoring in the cloud

the cloud, with asynchronous event sources. To achieve this goal, we decided to extend the *LOLA* specification langauge, combining concepts from Runtime Verification, Complex Event Processing and Time Series Databases.

The result is the specification language *Striver* [52], which is described in detail in Chapter 3.

Chapter 2

Preliminaries of Stream Runtime Verification

2.1 Temporal Logics

As mentioned in the Introduction, the correctness of computational systems often cannot be described in terms of their input and output, but instead it may depend on the execution trace of the system itself. For example, we may want to describe that a program is correct if a certain proposition holds *always* along its execution; or if at some point, *eventually*, a proposition is observed to be true.

One of the ways to reason about the execution of a program, is to think of it as a state transition system, in which atomic propositions have a truth value on each state. In this setup, properties about program execution can be expressed using temporal logics, including Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [11]. These logics were originally developed by philosophers for investigating how time is used in natural language, but their application to reasoning about computer programs for first used by Pnueli in the '70s.
In temporal logics, a program is described as a state transition system, and a trace of a program is an infinite list of such states.

In branching structure temporal logics, such as CTL, a logician can describe properties to deal with all the possible successors of a given state, this is, all the possible traces starting on the given state. For instance, if at a certain point the program under analysis can transition to more than one state, we can express that a property is violated if any of the successors does not satisfy a specific requirement.

On the other hand, in linear structure temporal logics, such as LTL, properties are described over a single trace, and thus, states have exactly one successor. For example, we can express that a property holds in a state if its only successor satisfies a certain requirement.

In Model Checking, a temporal logic formula holds for a transition system if it holds for all its potential traces. In Runtime Verification, we only care about one particular trace: the one we are observing. This is why RV dismisses branching structures like CTL and reuses the syntax and semantics for LTL.

The syntax of an LTL formula over the set AP of atomic propositions as described in [14] is the following:

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

Where $a \in AP$.

Given a trace $\sigma = s_0, s_1, ...$ of states such that for each state s and atomic proposition $a \in AP$, $s(a)$ indicates whether a holds in s, and $\sigma[j...] = s_j, s_{j+1}, ...$ is the suffix of σ starting from the $(j+1)$th position, the semantics of LTL formulas are defined recursively as follows:

$\sigma \vDash true$	always
$\sigma \vDash a$	iff $s_0(a)$
$\sigma \vDash \varphi_1 \wedge \varphi_2$	iff $\sigma \vDash \varphi_1$ and $\sigma \vDash \varphi_2$
$\sigma \vDash \neg\varphi$	iff $\sigma \nvDash \varphi$
$\sigma \vDash \bigcirc\varphi$	iff $\sigma[1...] \vDash \varphi$
$\sigma \vDash \varphi_1 \mathbf{U} \varphi_2$	iff $\exists j \geq 0. \sigma[j...] \vDash \varphi_2$ and $\sigma[i...] \vDash \varphi_1$ for all $0 \leq i < j$

There are two operators widely used in temporal logic formulas, which can be derived from the previous definitions.

The first of them is the "eventually" operator, represented by a $\Diamond$ symbol, which holds iff the accompanying property is satisfied either now or at some point in the future of the trace. For example, the formula $\Diamond a$ holds iff either now or further in the trace, the system will reach a state where a holds.

The second operator is the "forever" operator, represented by a $\Box$ symbol, which holds iff the accompanying property is satisfied both now and for the rest of the states in the future of the trace. For example, the formula $\Box a$ holds iff a holds in the current state and a also holds in every state further in the trace.

These operators can be defined using LTL in the following way:

$$\Diamond\varphi \stackrel{\text{def}}{=} true \mathbf{U} \varphi$$
$$\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$$

To examplify the behaviour of $\Diamond a$ and $\Box a$, consider the following trace:

instant	0	1	2	3	4	5	6	7	8
a	$false$	$false$	$false$	$false$	$true$	$false$	$false$	$true$	$true$
$\Diamond a$	$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$
$\Box a$	$false$	$false$	$false$	$false$	$false$	$false$	$false$	$true$	$true$

Since we observe a **true** value at time 8 in a, then the value of $\Diamond a$ is **true** from 0 to 8 (because a will *eventually* be **true**). On the other hand, since we observe **true** values at times 7 and 8 in a, then the value of $\Box a$ is **true** starting at time 7, because from that point on, it is true that *a always* holds.

Classic LTL formulas refer to future states, so in general they cannot be monitored online. LTL can be extended to define operators analogous to $\bigcirc$ and **U**, which refer to states in the past of the current state in the observed trace. The resulting formalism is called *Past-Linear Temporal Logic* (Past-LTL).

The analogous operator for **U** (called the "Until" operator) is **S** (called the "Since" operator), and we say that $\varphi_1 \mathbf{S} \varphi_2$ holds at a certain state s_i iff there is a previous state s_j in the trace such that φ_2 is satisfied in s_j and φ_1 is satisfied in every state $s_{j+1}, \ldots, s_i$.

Since traces are infinitely long, the semantics of $\bigcirc$ can refer safely to "the next state", because such state will always exist. This is not the case for its analogous operators, which refer to "the previous state": if the current state is the first state, then there is no such thing as a previous state. This is why we need to define two analogous operators to $\bigcirc$: the operator $\ominus$ holds for a state s_i iff it is not the first state in the trace (this is, if $i \neq 0$), and the accompanying property holds in s_{i-1}. On the other hand, the operator $\circledcirc$ holds for a state s_i iff either it is the first state (this is, if $i = 0$) or the accompanying property holds in s_{i-1}.

The analogous operator to $\Diamond$ in Past-LTL is $\Diamond\!\!\!-$. The property $\Diamond\!\!\!- a$ holds at a certain state if a has ever been witnessed to hold.

Finally, the analogous operator to $\Box$ in Past-LTL is $\boxminus$, and we say that a property $\boxminus a$ holds at a certain state if a holds at that state and all the states since the beginning of the trace.

Past-LTL properties can also be monitored online, computing the results while the program under scrutiny generates its trace. Given a Past-LTL formula and a trace of states, we can synthesize a program to check if the property holds for such trace.

However, the expressiveness of LTL is restricted to the boolean domain. Also, LTL formulas describe a logic predicate, so their definition and understanding require a logic mindset, which is not the usual way of thinking of engineers in the industry.

2.2 *LOLA*

LOLA is a specification language and algorithms for the online and offline monitoring of synchronous systems, which describes the computation of output streams from a given set of input streams, all of which advance at the same pace and have one value at every instant determined by a global clock. Specifications in *LOLA* resemble a declarative programming language, which helps improve the learning curve for programmers, and at the same time extends the expressiveness of specifications to richer domains, overcoming the limitation of the boolean domain restriction of LTL.

A *LOLA* specification is a set of equations over typed stream variables, of the form:

$$\begin{aligned} s_1 = &\quad e_1(t_1, \ldots, t_m, s_1, \ldots, s_n) \\ \vdots &\quad \vdots \\ s_n = &\quad e_n(t_1, \ldots, t_m, s_1, \ldots, s_n) \end{aligned}$$

Where $s_1, \ldots, s_n$ are the dependent variables, $t_1, \ldots, t_m$ are the independent variables and $e_1, \ldots, e_n$ are stream expressions, which are constructed as follows:

- If c is a constant of type T, then c is an atomic stream expression of type T.
- If s is a stream variable of type T, then s is an atomic stream expression of type T.
- Let $f : T_1 \times T_2 \times \cdots \times T_k \rightarrow T$ be a k-ary operator. If for $1 \leq i \leq k$, e_i is an expression of type T_i, then $f(e_1, \ldots, e_k)$ is a stream expression of type T.
- If b is a boolean stream expression and e_1, e_2 are stream expressions of type T, then $ite(b, e1, e2)$ is a stream expression of type T; note that ite abbreviates $if - then - else$.
- If e is a stream expression of type T, c is a constant of type T, and i is an integer, then $e[i, c]$ is a stream expression of type T. Informally, $e[i, c]$ refers to the value of the expression e offset i positions from the current position. The constant c indicates the default value to be provided in case an offset of i takes us past the end or before the beginning of the stream.

The semantics of *LOLA* specifications is defined in terms of evaluation models, which describe the relation between input streams and output streams.

Let φ be a *LOLA* specification over independent variables $t_1, \ldots, t_m$ with types $T_1, \ldots, T_m$ and dependent variables $s_1, \ldots, s_n$ with types $T_{m+1}, \ldots, T_{m+n}$ Let $\tau_1, \ldots, \tau_m$ be streams of length $N+1$, with τ_i of type T_i. A *valuation* is a tuple $(\sigma_1, \ldots, \sigma_n)$ of streams of length $N+1$ with appropriate types. We say that a valuation is an *evaluation model*, if for each equation in φ,

$$s_i = e_i(t_1, \ldots, t_m, s_1, \ldots, s_n)$$

$(\sigma_1, \ldots, \sigma_n)$ satisfies the following associated equations:

$$\sigma_i(j) = val(e_i)(j) \text{ for } 0 \leq j \leq N$$

where $val(e)(j)$ is defined as follows. For the base cases:

$$\begin{aligned} val(c)(j) &= c \\ val(t_i)(j) &= \tau_i(j) \\ val(s_i)(j) &= \sigma_i(j) \end{aligned}$$

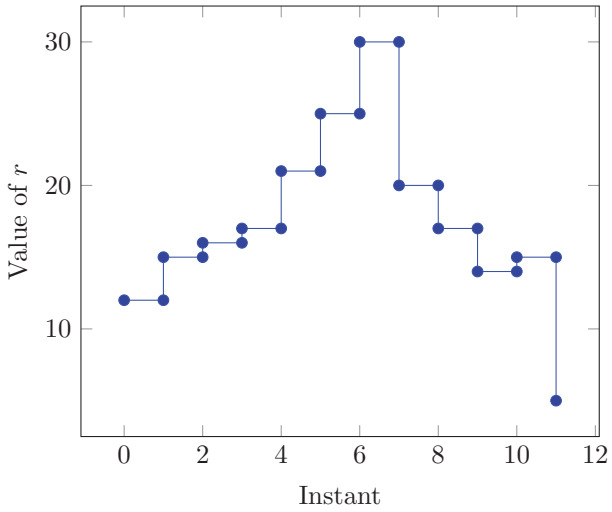


Figure 2.1: Signal r with synchronous streams

For the inductive cases:

$$val(f(e_1,\ldots,e_k)(j) = f(val(e_1)(j),\ldots,val(e_k)(j))$$

$$val(ite(b,e1,e2))(j) = if\ val(b)(j)\ then\ val(e_1)(j)\ else\ val(e_2)(j)$$

$$val(e[k,c])(j) = \begin{cases} val(e)(j+k) & \text{if } 0 \le j+k \le N \\ c & \text{otherwise} \end{cases}$$

Consider for example an input stream r of integers ranging from 0 to 100, which represents the use percentage of the memory at every instant (a stream of samples of the signal showing the load of the memory). In a particular run, the values of the signal could be the following:

instant	0	1	2	3	4	5	6	7	8	9	10	11
r	12	15	16	17	21	25	30	20	17	14	15	5

whose graphical representation is the piece-wise constant function shown in Figure 2.1.

In Past-LTL we can specify a specification s that captures whether the value of r has ever been higher than a certain threshold, for example 25%, using the following definition:

$$s = \diamondsuit r > 25$$

Using *LOLA*, we can specify an output signal s:

$$s = s[-1|\textbf{false}] \vee r > 25$$

The value of s at a given instant **t** is the disjunction between the value of s in the previous instant and the truth value of the fact that r is greater than 25 at **t**. At the first instant, the default value taken for $s[-1]$ is **false**.

Continuing with the previous example, the values of both streams would be the following:

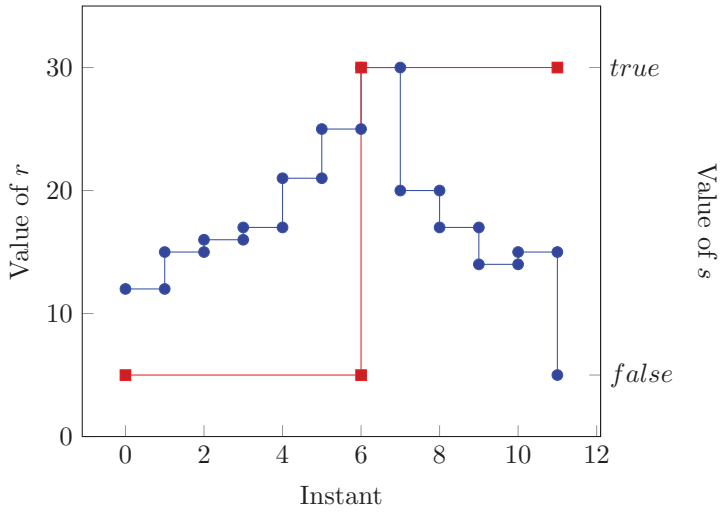


Figure 2.2: Signal s with synchronous streams

instant	0	1	2	3	4	5	6	7	8	9	10	11
r	12	15	16	17	21	25	30	20	17	14	15	5
s	**f**	**f**	**f**	**f**	**f**	**f**	**t**	**t**	**t**	**t**	**t**	**t**

The graphical representation of signals r and s is shown in Figure 2.2, where we can see how s transitions from **false** to **true** as soon as r's value becomes 30 at instant time 6 and remains **true** from that point onwards.

However, the syntax of *LOLA* is too permissive in the sense that it allows writing specifications where more than one valuation are evaluation models. For example, consider the specification φ:

$$s = s$$

with $s :: \mathbb{B}$. The valuation

instant	0	1	2	3	4	5	6	7	8	9	10	11
s	**f**	**f**	**f**	**f**	**f**	**f**	**t**	**t**	**t**	**t**	**t**	**t**

is an evaluation model of φ, but the valuation

instant	0	1	2	3	4	5	6	7	8	9	10	11
s	**t**	**f**	**f**	**t**	**t**	**t**	**f**	**t**	**t**	**f**	**t**	**f**

is a valuation model too.

In the same way, it can be the case that there is no valuation which is an evaluation model of a specification. Consider for example φ:

$$s = \neg s$$

with $s :: \mathbb{B}$. It is easy to see that there is no valuation where s is the negation of itself.

When a specification has more than one evaluation model, we say that the specification is *underdefined*. When a specification has no evaluation model, we say that the specification is *overdefined*.

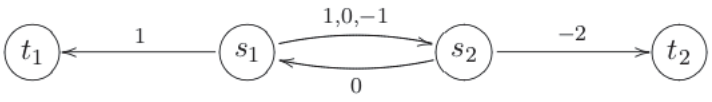


Figure 2.3: Dependency graph of the specification in Example 1

And when a specification has exactly one valuation which is an evaluation model of it, we say that the specification is *well-defined*.

The condition of well-definedness is a semantic condition, which is not easy to check for a given specification. In fact, it is easy to show that for rich enough domains, deciding whether a given specification is well-defined is undecidable. We develop here a syntactic condition, called *well-formedness*, that is easy to check on input specifications and guarantees that specifications are well-defined.

The well-formedness of a specification is checked by analysing its *dependency graph*, which is defined in the following way:

Let φ be a *LOLA* specification. A dependency graph for φ is a weighted and directed multi-graph $G = \langle V, E \rangle$, with vertex set $V = \{s_1, \ldots, s_n, t_1, \ldots, t_m\}$.

An edge $e : \langle s_i, s_k, w \rangle$ labeled with a weight w is in E iff the equation for $\sigma_i(j)$ in φ_σ contains $\sigma_k(j + w)$ as a subexpression of the right-hand side, for some j (or $e : \langle s_i, t_k, w \rangle$ for subexpression $\tau_k(j + w)$).

Intuitively, the edge records the fact that s_i at a particular position depends on the value of s_k, offset by w positions. Note that there can be multiple edges between s_i and s_k with different weights on each edge. Vertices labeled by t_i do not have outgoing edges.

Take for example the following specification:

Example 1.

$$s_1 = s_2[1,0] + s_2 + s_2[-1,0] + t_1[1,0]$$
$$s_2 = s_1 + t_2[-2,1]$$

Its dependency graph, shown in Figure 2.3, has three edges from s_1 to s_2, with weights 1, 0, -1, and one zero weighted edge from s_2 back to s_1. There is one edge from s_1 to t_1, and one from s_2 to t_2. A walk of a graph is a sequence $v_1, \ldots, v_{k+1}$ of vertices, for $k \geq 1$, and edges $e_1, \ldots, e_k$, such that $e_i : \langle v_i, v_{i+1}, w_i \rangle$. The walk is closed iff $v_1 = v_{k+1}$. The total weight of a walk is the sum of weights of its edges.

Definition A *LOLA* specification is well-formed if there is no closed-walk with total weight zero in its dependency graph.

LOLA has been proved successful due to its robustness and succinct specifications, which are straightforward and easy to understand; without detriment to expressivity. The formal syntax and semantics of the language can be found in [32].

In certain scenarios, however, it is inaccurate to assume a global clock and signals to have a value at each instant. Such is the case of input streams which sample their corresponding signals at different rates, or an input stream reporting the points of interest a running program is going through.

If we consider streams of timestamped values, we need a model to deal with them and perform an *Asynchronous Stream Runtime Verification*. In this setup, streams have not only a value, but also the instant at which such value was generated. In other words, for a time domain $\mathbb{T}$, every element of a stream of type $\mathbb{D}$ is a pair (t,d) such that $t \in \mathbb{T}$ and $d \in \mathbb{D}$.

The semantics of a *LOLA* specification are unclear for this kind of streams. Take for example the definition of signal $s = r[-1|0]$. What does it mean to refer to $r[-1]$? Is it the last witnessed value of r or is it the value of r when the last event of s was witnessed?

We extend *LOLA* to work with asynchronous streams and solve these ambiguities in a new formalism called *Striver* [52], a Stream Runtime Verification language for the monitoring of asynchronous systems, whose details are presented in Chapter 3

Chapter 3

Striver

In this chapter we present *Striver*, a Stream Runtime Verification formalism (hence its name, STReam Verification), which targets the runtime verification of asynchronous, distributed systems by borrowing concepts from Runtime Verification, Complex Event Processing and Time Series Databases; and combining them into a langauge simple enough to be studied with mathematical rigor, and as a consequence enabling the resource boundaries of a specification in terms of memory and response time.

Essentially, *Striver* is a specification formalism for timed asynchronous observations, where streams are sequences of timed events, not necessarily happening at the same time in all input streams, but where all time-stamps are totally ordered according to a global clock (following the timed asynchronous model of distributed systems [53]).

The monitor is not intertwined nor modifies the system under analysis, but instead runs on its specific infrastructure, with the goal of not affecting the system's behavior (non-intrusiveness). See [54] for a definition and classification of these concepts. The target practical system is the monitoring and testing of cloud applications, where this assumption is reasonable.

In Section 3.1, we give an introduction to *Striver* following the example presented on the preliminaries. Section 3.2 contains the formal definition of preliminary concepts for *Striver*. In Section 3.3, we describe the syntax of the specification language. In Section 3.4, we present the denotational semantics of a specification. In Section 3.5, we describe what it means for a specification to be well-defined, and give a well-formedness condition over a specification, which is sufficient to state that it is well-defined. Section 3.6 contains an online algorithm, proves its correctness and shows that the algorithm is trace length independent. Section 3.7 reports on an empirical evaluation of the prototype implementation of *Striver*. In Section 3.8, we show that *Striver* can be used as a low level language to compile *TeSSLa* [35]. Finally, in Section 3.9 we present a real world use case where *Striver* has been successfully applied.

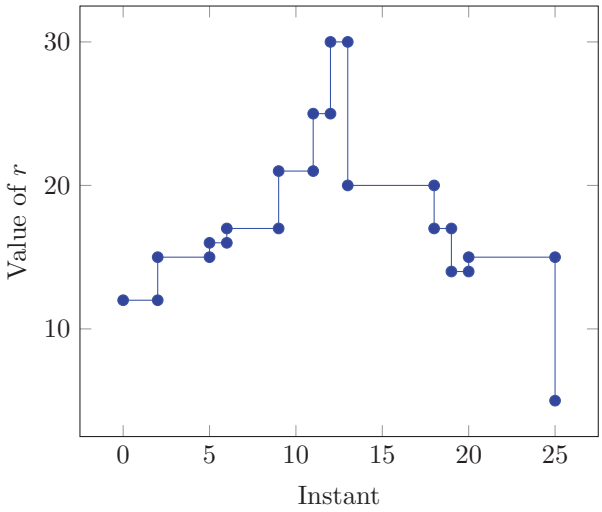


Figure 3.1: Signal r with asynchronous streams

3.1 Introduction

Just like *LOLA* specifications, *Striver* specifications resemble a declarative programming language and are based on the description of the computation of output streams from a given set of input streams. However, in *Striver*, events are stamped with the time at which they were generated, and may be produced at different points in time for every source as long as the timestamps are monotonically increasing for each input stream.

Since the instants at which streams emit events are no longer fixed by a global clock, to define an output stream using *Striver*, it is necessary to specify the values of the output streams — just like in *LOLA*— but also the times at which these values are emitted.

The set of times at which an event is emitted over a stream $x : \mathbb{D}$ are called the *ticks* of x, and for each $t : \mathbb{T}$ in this set, there is a value for signal x at that point, which is noted $x(t) : \mathbb{D}$.

In *LOLA* specifications, we can naturally think of a stream of type D as an array of values of type D, indexed by the instant at which each of them was produced. In *Striver*, instead, we can naturally think of a stream of type D as an array of pairs $(\mathbb{T}, D)$, where the index is meaningless, and the first component of each pair indicates the instant at which the value on its second component is produced.

Following the example introduced in Chapter 2, consider an input stream r of integers ranging from 0 to 100, which represents the use percentage of the memory at the moments of sampling. In a particular run, the values of the signal could be the following:

instant	0	2	5	6	9	11	12	13	18	19	20	25
r	12	15	16	17	21	25	30	20	17	14	15	5

whose graphical representation is shown in Figure 3.1.

Using *Striver*, the specification of an output signal s to check if the value of r has ever been higher

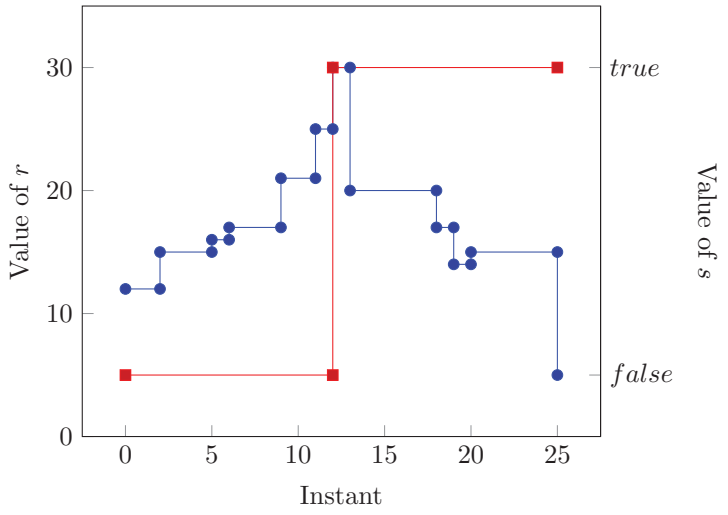


Figure 3.2: Signal s with asynchronous streams

than 25 percent would be the following:

$$s.ticks = r.ticks$$
$$s(t) = s(< t | false) \vee r(\sim t) > 25$$

This specification states that the set of points at which s will emit an event are the points at which an event is received over r (since this is the definition of the *ticks* of s); and the value of s at those instants is the disjunction between the last value of s and the truth value of the fact that r is greater than 25 at that instant (this is the definition of $s(t)$ for each t in the *icks* of s). Also, when the first event of r is witnessed, and thus "the last value of s" does not exist, the default value taken for $s(< t)$ is **false**. Notice the similarity between the *LOLA* specification of the example presented in Chapter 2

$$s = s[-1|false] \vee r > 25$$

and the value expression of the *Striver* specification

$$s(t) = s(< t | false) \vee r(\sim t) > 25$$

Continuing with the previous example, the values of both streams would be the following:

instant	0	2	5	6	9	11	12	13	18	19	20	25
r	12	15	16	17	21	25	30	20	17	14	15	5
s	**f**	**f**	**f**	**f**	**f**	**f**	**t**	**t**	**t**	**t**	**t**	**t**

The graphic of signals r and s is shown in Figure 3.2, where it can be seen how s transitions from **false** to **true** as soon as r's value becomes 30 at time 12, and remains **true** from that point onwards. This example illustrates the general idea of a *Striver* specification, showing that *Striver* can be thought of as an extension of *LOLA* to work with dense time domains.

We start by introducing a version of *Striver* that allows defining efficiently monitorable specifications [32], those for which all streams can be resolved immediately. Section 3.6 contains an online

monitoring algorithm and the proof that this algorithm is also trace length independent. But first, we need to present some preliminary definitions to construct the semantics of *Striver* over them.

3.2 Striver preliminaries

The main idea behind Stream Runtime Verification is to separate two concerns: the temporal dependencies and the data manipulated, for which we use data domains.

Data Domains.

We use many-sorted first order logic to describe data domains. The expressions sort and type are used interchangeably in the rest of the thesis.

A simple theory, *Booleans*, has only one sort, *Bool*, two constants `true` and `false`, binary functions $\wedge$ and $\vee$, unary function $\neg$, etc.

A more sophisticated signature is *Naturals* that consists of two sorts (*Nat* and *Bool*), with constant symbols `0`, `1`, `2`... of sort *Nat*, binary symbols $+$, $*$, etc (of sort $Nat \times Nat \to Nat$) as well as predicates $<$, $\leq$, etc of sort $Nat \times Nat \to Bool$, with their usual interpretation.

All theories have equality and are typically (e.g. *Naturals*, *Booleans*, *Queues*, *Stacks*, etc) equipped with a ternary symbol `if·then·else·`. In the case of *Naturals*, the `if·then·else·` symbol has sort $Bool \times Nat \times Nat \to Nat$.

For example, a functional symbol `f` that describes a function that takes values from S_1 and S_2 and returns a value in S_{res} has type $S_1 \times S_2 \to S_{\text{res}}$, where S_1, S_2 and S_{res} are sorts in $\mathscr{S}$. The arity of a function is the number of its arguments, each of the appropriate sort. In the example above, `f` has arity 2. Functions of arity 0 represent constants. Examples of sorts are shown in Fig. 3.3.

Our theories are interpreted, so each sort S is associated with a domain D_S (a concrete set of values), and each function symbol `f` is interpreted as a total computable function f, with the given arity and that produces values of the domain of the result given elements of the arguments' domains. We omit the sort S when it is clear from the context.

We will use *stream variables* with an associated sort, but from the point of view of the theories, these stream variables are atoms. As usual, given a set of sorted atoms A and a theory, terms are the smallest set containing A and closed under the use of function symbol in the theory as a constructor (respecting sorts).

We consider a special *time* domain $\mathbb{T}$, whose interpretation is a (possibly infinite, possibly dense) set with a total order and a minimal element 0, and a binary addition symbol $+$. Examples of time domains are $\mathbb{R}_0^+$, $\mathbb{Q}_0^+$ and $\mathbb{N}_0$ with their usual order. Given $t_a, t_b \in \mathbb{T}$ we use $[t_a, t_b] = \{t \in \mathbb{T} \mid t_a \leq t \leq t_b\}$, and also (t_a, t_b), $[t_a, t_b)$ and $(t_a, t_b]$ with the usual meaning. We say that a set of time points $S \subseteq \mathbb{T}$ does not contain bounded infinite subsets, whenever for every $t_a, t_b \in \mathbb{T}$, the set $S \cap [t_a, t_b]$ is finite, in which case we say that S is a non-Zeno set.

Name: *Booleans*	Name: *Naturals*
Sorts: *Bool*	Sorts: *Nat, Bool*
Symbols: $\texttt{true}, \texttt{false} : \mathit{Bool}$ $\texttt{and}, \texttt{or} : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}$ $\texttt{not} : \mathit{Bool} \to \mathit{Bool}$	Symbols: $0, 1, 2, \ldots : \mathit{Nat}$ $+, * : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$ $\begin{bmatrix}\texttt{if}\cdot\texttt{then}\cdot\\ \texttt{else}\cdot\end{bmatrix} : \mathit{Bool} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$
Name: *Stacks*	Name: *Sets*
Sorts: *Stack, Bool, Elem*	Sorts: *Set, Bool, Elem*
Symbols: $\texttt{emp} : \mathit{Stack}$ $\texttt{push} : \mathit{Elem} \times \mathit{Stack} \to \mathit{Stack}$ $\texttt{pop} : \mathit{Stack} \to \mathit{Stack}$ $\texttt{top} : \mathit{Stack} \to \mathit{Elem}$ $\begin{bmatrix}\texttt{if}\cdot\texttt{then}\cdot\\ \texttt{else}\cdot\end{bmatrix} : \mathit{Bool} \times \mathit{Stack} \times \mathit{Stack} \to \mathit{Stack}$	Symbols: $\texttt{empty} : \mathit{Set}$ $\texttt{union} : \mathit{Set} \times \mathit{Set} \to \mathit{Set}$ $\texttt{inters} : \mathit{Set} \times \mathit{Set} \to \mathit{Set}$ $\texttt{setdiff} : \mathit{Set} \times \mathit{Set} \to \mathit{Set}$ $\texttt{single} : \mathit{Elem} \to \mathit{Set}$ $\texttt{add} : \mathit{Elem} \times \mathit{Set} \to \mathit{Set}$ $\begin{bmatrix}\texttt{if}\cdot\texttt{then}\cdot\\ \texttt{else}\cdot\end{bmatrix} : \mathit{Bool} \times \mathit{Set} \times \mathit{Set} \to \mathit{Set}$

Figure 3.3: Examples of domains: *Booleans*, *Naturals*, *Stacks*, *Sets*

We extend every domain D into $D^{\perp}$ that includes two special fresh symbols $\perp^{\mathbb{D}}_{\text{outside}}$ and $\perp^{\mathbb{D}}_{\text{notick}}$. These new symbols allow capturing when a stream does not generate an event, and when the time offset falls off the beginning and the end of the trace.

Streams.

Monitors observe sequences of events as inputs, where each event is time-stamped and contains a data value from its domain.

Event stream An event stream of sort D is a partial function $\eta : \mathbb{T} \rightharpoonup D$ such that $dom(\eta)$ does not contain bounded infinite subsets, where $dom(\eta)$ is the subset of $\mathbb{T}$ where f is defined.

The set $dom(\eta)$ is called the set of *event points* of η. An event stream η can be naturally represented as a *timed word*: $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (E(\eta) \times D)^*$, or as an ω-timed word $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (E(\eta) \times D)^\omega$ for infinite streams, such that:

1. s_η is ordered by time ($t_i < t_{i+1}$); and
2. for every $t_a, t_b \in \mathbb{T}$ the set $\{(t, d) \in s_\eta \mid t \in [t_a, t_b]\}$ is finite.

The set of all event streams over D is denoted by $\mathscr{E}_D$.

We introduce some notation for event streams. The functions $prev_<$ and $prev_\leq$ with type $\mathscr{E}_D \times \mathbb{T} \to \mathbb{T}^{\perp}$ are defined as follows. Note that the functions can return a value in $\mathbb{T}^{\perp}$ because *sup* can return

$\bot^{\mathbb{T}}_{\text{outside}}$ when the stream has no event in the interval provided.

$$prev_<(\sigma, t) \stackrel{\text{def}}{=} sup(dom(\sigma) \cap [0, t)) \qquad prev_\leq(\sigma, t) \stackrel{\text{def}}{=} sup(dom(\sigma) \cap [0, t])$$

$$sup(S) \stackrel{\text{def}}{=} \begin{cases} max(S) & \text{if } S \neq \emptyset \\ \bot^{\mathbb{T}}_{\text{outside}} & \text{otherwise} \end{cases}$$

Essentially, given a stream σ and a time instant $t \in \mathbb{T}$, the expression $prev_<(\sigma, t)$ provides the nearest time instant in the past of t at which σ is defined. Similarly, $prev_\leq(\sigma, t)$ returns t if $t \in dom(\sigma)$, otherwise it behaves as $prev_<$.

Synchronous SRV

In synchronous SRV formalisms, like *LOLA*, specifications are described by associating every output stream variable y with a defining equation that, once the input streams are known, associates y to an output stream. For example:

```
define bool always_p := p /\ always_p[-1,true]
define int   count_p := (count_p[-1,0]) + if p then 1 else 0
```

defines two output streams: `always_p`, which calculates whether Boolean input stream p was true at every point in the past (that is, $\boxminus p$) and `count_p`, which counts the number of times p was true in the past.

Offset expressions like `count_p[-1,0]` allow referring to streams in a different position (in this case in the previous position) with a default value when there is no previous position (the beginning of the trace).

In this thesis we introduce a similar formalism for timed event streams. Our goal is to provide a simple language with few constructs including explicit references to the previous position at which some stream contains an event, contrary to other stream languages like *TeSSLa* [35] and RTLola [36] which preclude to reason about real-time instants. We say that *Striver* is an *explicit time* SRV formalism.

3.3 Syntax

A *Striver* specification describes the relation between input event-streams and output event-streams, where an input stream is a sequence of observations from the system under analysis.

The key idea in *Striver* is to define for each defined stream variable:

- a *ticking expression* that captures when the stream may contain an event;
- a *value expression* that defines the value contained in the event.

Note that in synchronous SRV (like in the example above), only a value expression is necessary because every stream has a value at every clock tick.

Formally, a *Striver* specification $\varphi : \langle I, O, V, T \rangle$ consists of input stream variables $I = \{x_1, \ldots, x_n\}$, output stream variables $O = \{y_1, \ldots, y_m\}$, a collection of ticking expressions $T = \{T_1, \ldots, T_m\}$ and a collection of value expressions $V = \{V_1, \ldots, V_m\}$. For output variable y, T_y captures when stream y ticks and V_y what the value is when it ticks.

All input and output streams are associated with a sort. It is sometimes convenient to partition output streams into proper outputs and intermediate streams, that are introduced only to simplify specifications.

In practice, it is very useful that T_y defines an over-approximation of the set of instants at which y ticks, and then allow the value expression to evaluate to $\bot^{\mathbb{D}}_{\text{notick}}$. The stream associated with y does not contain an event at t if V_y evaluates to $\bot^{\mathbb{D}}_{\text{notick}}$ at t, even if t is in T_y.

For example, if one wishes y to filter out events from a given stream x it is simple to define in T_y that y ticks whenever x does, and delegate to V_y to decide whether an event is relevant or should be filtered out.

Example 2. Consider the following specification:

```
input int x

ticks y          := x.ticks
define int y t   := if x%2==0
                    then x(~t)
                    else notick
```

The stream y filters out the odd values of x. □

Expressions.

We fix a set of stream variables $Z = I \cup O$. Apart from ticking expressions and value expressions, offset expressions (used inside value expressions) allow defining temporal dependencies between ticking instants.

- *Ticking Expressions*:

$$\alpha := \{c\} \mid v.\textbf{ticks} \mid \alpha \cup \alpha \mid \textbf{delay}\ w$$

  where $c \in \mathit{time}$ is a time constant, v is an arbitrary stream variable, and w is a stream variable of type $\mathbb{T}_\epsilon$.

  The type $\mathbb{T}_\epsilon$ is defined as $\mathbb{T}_\epsilon = \{t \mid t \geq \epsilon\}$ for a given $\epsilon > 0$. This restriction on the argument of **delay** guarantees that the ticking instants are non-zeno if all their inputs are non-zeno, as is proved in Section 3.6.

- *Offset Expressions*, which allow fetching previous events from streams:
$$\tau_x ::= x <\!\!\sim \tau' \mid x \ll \tau' \qquad \tau' ::= \mathbf{t} \mid \tau_z \text{ for } z \in Z$$
Offset expressions have sort $\mathbb{T}^{\perp}$.

  Here, **t** represents the current value of the clock.

  The intended meaning of $x \ll e$ is to refer to the previous instant strictly in the past of e where x ticks (or $\perp^{\mathbb{T}}_{\text{outside}}$ if there is not such an instant).

  The expression $x <\!\!\sim e$ also considers the present.

- *Value Expressions*, which give the value of a defined stream at a given ticking point candidate:
$$E ::= d \mid x(\tau_x) \mid \mathbf{f}(E_1, \ldots, E_k) \mid \mathbf{t} \mid \tau_x \mid \mathbf{outside}_{\mathbb{D}} \mid \mathbf{notick}_{\mathbb{D}}$$
where d is a constant of type D, $x \in Z$ is a stream variable of type D and **f** is a function symbol of return type D.

  Note that in $x(\tau_x)$ the value of stream x is fetched at an offset expression indexed by x, which captures the ticking points of x, which guarantee the existence of an event.

  Expressions **t** and τ_x are included to build expressions of sort $\mathbb{T}$.

  The two additional constants $\mathbf{outside}_{\mathbb{D}}$ and $\mathbf{notick}_{\mathbb{D}}$ allow to reason about accessing the end of the streams, or not generating an event at a ticking candidate instant.

  Notice that we do not allow fetching the value of a stream at an arbitrary point in time. Streams are accessed "jumping" between the ticking instants of other streams or its own. This restriction enables the trace-length independent monitorization of a specification.

We also use the following syntactic sugar:

$$x(\tilde{}e) \stackrel{\text{def}}{=} x(x<\!\!\sim e) \qquad x(\tilde{}e, d) \stackrel{\text{def}}{=} \texttt{if } (x <\!\!\sim e)\texttt{==outside then } d \texttt{ else } x(\tilde{}e)$$
$$x(<e) \stackrel{\text{def}}{=} x(x \ll e) \qquad x(<e, d) \stackrel{\text{def}}{=} \texttt{if } (x \ll e)\texttt{==outside then } d \texttt{ else } x(<e)$$

Essentially, $x(\tilde{}\mathbf{t})$ provides the value of x at the previous ticking instant of x (including the present) and $x(<\mathbf{t})$ is similar but not including the present.

Also, $x(<\mathbf{t}, d)$ is the analogous to $x[-1, d]$ in synchronous SRV, allowing to fetch the value in the previous event in stream x, or d if there is not such a previous event.

Example 3. If we have an input stream **sale** that represents sales of a certain product, and an input stream **arrival** which represents the arrivals of the same product to the store, we can define a signal **stock** to compute the stock of this product using the following specification:

```
input int sale, int arrival
ticks stock := sale.ticks U arrival.ticks
define int stock t := stock(<t,0) +
    (if isticking(arrival) then arrival(~t) else 0) -
    (if isticking( sale )  then  sale( ~t ) else 0)
```

The macro `isticking` is defined for an arbitrary stream *s*in the following way:

```
macro isticking(s) := s(<~t) == t
```

Note that **stock** is defined to tick when either **sale** or **arrival** (or both) tick, since these are the only points at which the stock of the product might change. □

Example 4. To illustrate the use of **delay** consider the following specification:

```
ticks clock        := {0} U delay clock
define Time_eps clock t := 1sec
```

The stream `clock` emits an event every second since time 0. □

3.4 Denotational semantics

The semantics of *Striver* is defined denotationally, establishing whether a given input and a given output satisfy the specification, which is defined in terms of *valuations*.

Given a set of variables Z, a valuation σ is a map that associates every x in Z of sort D with an event stream from $\mathscr{E}_D$. Given a valuation σ we define the result of evaluating an expression for σ. We define three evaluation maps $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$, $[\![.]\!]_\sigma$ depending on the type of the expression:

- *Ticking Expressions.* The semantic map $[\![.]\!]_\sigma$ assigns a set of time instants to each ticking expression as follows:

$$\begin{aligned}
[\![\{c\}]\!]_\sigma &\stackrel{\text{def}}{=} \{c\} \\
[\![v.\mathbf{ticks}]\!]_\sigma &\stackrel{\text{def}}{=} dom(\sigma_v) \\
[\![a \cup b]\!]_\sigma &\stackrel{\text{def}}{=} [\![a]\!]_\sigma \cup [\![b]\!]_\sigma \\
[\![\mathbf{delay}(w)]\!]_\sigma &\stackrel{\text{def}}{=} \{t' \mid \text{ there is a } t \in dom(\sigma_w) \text{ such that } t + \sigma_w(t) = t' \\
&\qquad \text{and } dom(\sigma_w) \cap (t, t') = \emptyset \}
\end{aligned}$$

- *Offset Expressions.* For offset expressions $[\![.]\!]_\sigma$ provides, given a time instant t, another time instant[1]:

$$\begin{aligned}
[\![\mathbf{t}]\!]_\sigma(t) &\stackrel{\text{def}}{=} t \\
[\![x \ll e]\!]_\sigma(t) &\stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{T}}_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ prev_<(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases} \\
[\![x <\sim e]\!]_\sigma(t) &\stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{T}}_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ prev_\leq(\sigma_x, [\![e]\!]_\sigma(t)) & \text{otherwise} \end{cases}
\end{aligned}$$

[1] Notice that we consider $\bot^{\mathbb{T}}_{\text{outside}}$ to be a ticking instant.

- *Value Expressions.* Finally, value expressions are evaluated into event streams of the appropriate type. For a given instant t:

$$
\begin{aligned}
[\![d]\!]_\sigma(t) &\stackrel{\text{def}}{=} d \\
[\![x(e)]\!](t) &\stackrel{\text{def}}{=} \begin{cases} \bot^{\mathbb{D}}_{\text{outside}} & \text{if } [\![e]\!]_\sigma(t) = \bot^{\mathbb{T}}_{\text{outside}} \\ v & \text{if } [\![e]\!]_\sigma(t) = t' \text{ and } \sigma_x(t') = v \end{cases} \\
[\![\mathbf{f}(E_1,\ldots,E_k)]\!]_\sigma(t) &\stackrel{\text{def}}{=} f([\![E_1]\!]_\sigma(t),\ldots,[\![E_k]\!]_\sigma(t)) \\
[\![t_x]\!]_\sigma(t) &\stackrel{\text{def}}{=} [\![t_x]\!]_\sigma(t) \\
[\![\mathbf{outside}_{\mathbb{D}}]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot^{\mathbb{D}}_{\text{outside}} \\
[\![\mathbf{notick}_{\mathbb{D}}]\!]_\sigma(t) &\stackrel{\text{def}}{=} \bot^{\mathbb{D}}_{\text{notick}}
\end{aligned}
$$

  Note that $[\![x(e)]\!]_\sigma$ includes the possibility that

  1. the expression cannot be evaluated because the time instant given by $[\![e]\!]_\sigma(t)$ is outside the boundaries of domain of the stream, and
  2. the expression is not defined because the stream does not tick at t.

  It is easy to see that the cases for $[\![x(e)]\!]_\sigma$ are exhaustive because $[\![e]\!]_\sigma(t)$ guarantees that $\sigma_x(t')$ is defined.

For example, consider the following stream $(1.0, 17), (2.5, 21), (3.5, 12)$ for variable `sale` from Example 3. Then

$$[\![\texttt{sale(~t)}]\!]_\sigma(3.1) = [\![\texttt{sale(sale<~ t)}]\!]_\sigma(3.1) = [\![\texttt{sale}]\!]_\sigma(2.5) = 21$$

Evaluation Model Given a valuation σ of variables $I \cup O$ the evaluation of the equations for stream $y \in O$ is:

$$[\![T_y, V_y]\!] \stackrel{\text{def}}{=} \{(t,d) \mid t \in [\![T_y]\!]_\sigma \text{ and } d = [\![V_y]\!](t) \text{ and } d \neq \bot^{\mathbb{D}}_{\text{notick}}\}$$

An evaluation model is a valuation σ such that for every $y \in O$: $\sigma_y = [\![T_y, V_y]\!]$.

Just like in *LOLA*, *Striver* specifications can admit many evaluation models. Consider for example the following specification φ:

```
ticks x = x.ticks
define unit x t := ()
```

A valuation σ such that $\sigma_x = [(0,()),(15,()),(17,())]$ is an evaluation model of φ, but also a valuation σ'_x such that $\sigma_x = [(3,())]$, and $\sigma \neq \sigma'$. When a specification accepts many evaluation models, we say that the specification is *underdefined*.

It can also be the case that a specification admits no evaluation models. For example, for the following specification:

```
ticks x = {0}
define bool x t := !x(~t)
```

it is easy to see that no evaluation model will satisfy the fact that x produces both a boolean value and its negation at time 0. When a specification accepts no evaluation models, we say that the specification is *overdefined*.

We say that a specification is *well-defined* if it admits exactly one evaluation model, and thus we can think of the corresponding monitor as a computable function from input streams to output streams. The following definition captures whether a specification indeed corresponds to such a function.

Well-defined A specification φ is *well-defined* if for all σ_I, there is a unique σ_O, such that $\sigma_I \cup \sigma_O$ is an evaluation model of φ.

Additionally, a specification is efficiently monitorable if the output for time t only depends on the input for time t, which enables the incremental computation of the output stream.

Efficiently monitorable A well-defined specification φ is *efficiently monitorable* whenever for every two input σ_I and σ'_I with evaluation models σ_O and σ'_O, and for every time t, if $\sigma_I|_t = \sigma'_I|_t$ then $\sigma_O|_t = \sigma'_O|_t$.

3.5 Well-formedness

The condition of well-definedness is a semantic condition, which is not easy to check for a given specification. In fact, this condition is undecidable for expressive enough domains. We present here a syntactic condition, called well-formedness, that is easy to check on input specifications and guarantees that specifications are well-defined. All specifications encountered in practice are well-formed.

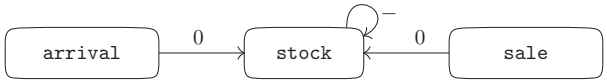
We first define a subset of the offset expressions, called the *Present* subset, as the smallest subset that contains **t** and such that if $e \in \mathit{Present}$ then $(x <\tilde{} \ e) \in \mathit{Present}$. We say that an output stream variable y directly depends on a stream variable x (and we write $x \to y$) if x appears in T_y or V_y. We say that y has a present direct dependency on x (and write $x \xrightarrow{0} y$) if $x \to y$ and either

- $x.\mathit{ticks}$ appears in T_y, or
- $(x <\tilde{} \ e)$ appears in V_y and $e \in \mathit{Present}$.

A direct dependency captures whether in order to compute a value of a stream variable y at position t, it is necessary to know the value of stream variable x up to t. If $x \to y$ but $x \not\xrightarrow{0} y$ we say that y directly depends on x in the past (and we write $x \xrightarrow{-} y$).

Dependency Graph The dependency graph of a specification φ is a graph (V, E) where $V = I \cup O$ and $E = V \times V \times \{\xrightarrow{0}, \xrightarrow{-}\}$.

The dependency graph of Example 3 which computes the stock of a certain product is:



The following definition captures whether an output stream variable cannot depend on itself at the present moment.

Well-Formed Specifications A specification φ is well-formed if every closed path in its dependency graph contains a past dependency edge $\xrightarrow{-}$.

Closed paths in the dependency graph correspond to dependencies between a stream and itself in the specification φ. These closed paths do not create problems if the path corresponds to accessing the strict past of the stream.

Note that if one removes $\xrightarrow{-}$ edges from the dependency graph of a well-formed specification, the resulting graph is necessarily a DAG. In other words $\xrightarrow{0}{}^*$ is irreflexive.

The following lemma formally captures the information that is sufficient to determine the value of a given stream at a given time instant.

Lemma 1. Let y be an output stream variable of a specification φ, σ, σ' be two evaluation models of φ, such that, for time instant t:

(i) For every variable x, $\sigma_x(t') = \sigma'_x(t')$ for every $t' < t$, and
(ii) For every x, such that $x \xrightarrow{0}{}^* y$, $\sigma_x(t') = \sigma'_x(t')$ for every $t' \leq t$

Then $\sigma_y(t) = \sigma'_y(t)$.

Proof. Note that since σ_y and σ'_y must satisfy that $T_y = [\![T_y]\!]_\sigma$ and $T_y = [\![T_y]\!]_{\sigma'}$, and also $V_y[\![V_y]\!]_\sigma$ and $V_y = [\![V_y]\!]_{\sigma'}$. It is easy to see that $t \in [\![T_y]\!]_\sigma$ if and only if $\in [\![T_y]\!]_{\sigma'}$, by structural induction on ticking expressions. The key observation is that only values in the conditions of the lemma are needed for the evaluation, which are assumed to be the same in σ and σ'. Similarly, it is easy to see that $[\![V_y]\!]_\sigma = [\![V_y]\!]_{\sigma'}$ because again the values needed are the same in σ and σ'. □

We are now ready to show that well-formed specifications cannot have two different evaluation models.

Theorem 1. Every well-formed *Striver* specification is well-defined.

Proof. Let φ be a well-formed specification and σ and σ' two evaluation models for the same input (that is $\sigma_I = \sigma'_I$). We show that $\sigma = \sigma'$. Since φ is well-formed, the dependency graph removing $\xrightarrow{-}$ is acyclic. Let $<$ be an arbitrary total order between stream variables such that if $x \xrightarrow{0}{}^* y$ then $x < y$. Note that $<$ is simply a reverse topological order if this acyclic graph. Now we derive a total order between the events occurring in streams in σ as follows. Let $(t, d) \in \sigma_x$ and $(t', d)' \in \sigma_y$ be two such events. Then

- $(t, d) \prec (t', d')$ whenever $t < t'$, or

Figure 3.4: Dependency graph of φ

- $(t,d) \prec (t',d')$ whenever $t = t'$ and $x < y$.

Since $<$ is a total order between variables, $\prec$ is also a total order between all events in σ. We now show by induction in the total order that $\sigma = \sigma'$. Consider the first event (t,d) in σ according to $\prec$. This event can be either:

- an input event. In this case, since $\sigma_I = \sigma'_I$, (t,d) must also be the first input event in σ'_I. The only possibility for σ' to differ in the first event is that some output stream y has an event $(t'd') \prec (t,d)$. But this is only possible if the defining equations for y with no previous events make $t' \in [\![T_y]\!]_\sigma$ and $d' = [\![V_y]\!]_\sigma$, but by Lemma 1 (t',d) is also an event in σ_y, which contradicts that (t,d) was the first event.
- an event in an output stream y. In this case, again $t \in [\![T_y]\!]_\sigma$ if and only if $t \in [\![T_y]\!]_{\sigma'}$ and $d = [\![V_y]\!]_\sigma = [\![V_y]\!]_{\sigma'}$.

Assume now the inductive hypothesis that both streams coincide up to the i-th event and consider the i+1 event in σ. Again, if the event (t,d) is an input event it must also by the following input event in σ', and no output event can precede it because it would also precede (t,d) in σ because the defining equations depend on the i-th preceding events. Also, if the event (t,d) is an output events, since the evaluations of the defining events are the same in σ and σ', (t,d) will also be an event in σ' (and cannot be preceded by another event). This finishes the proof. □

Notice that even though well-formedness implies well-definedness, the opposite is not true. Take for instance the following specification φ:

```
ticks x := x.ticks
define void x t := notick
```

φ is well-defined because the only evaluation model is that in which x emits no event, but the specification is not well-formed because its dependency graph, shown in Figure 3.4 contains a closed path with only $\xrightarrow{0}$ arcs: $x \xrightarrow{0} x$.

Consider the example 3 presented in Section 3.3, where we define a signal **stock** to compute the stock of a certain product based on its **sales** and **arrivals**:

```
input int sale, int arrival
ticks stock := sale.ticks U arrival.ticks
define int stock t := stock(<t,0) +
    (if isticking(arrival) then arrival(~t) else 0) -
    (if isticking( sale )  then  sale( ~t ) else 0)
```

Example 5. The following example defines a *Bool* stream to monitor whether the stock falls below a predefined `threshold`.

```
const threshold := 100
ticks low_stock := stock.ticks
define bool low_stock t := stock(~t) < threshold
```

We can use the stream variable `low_stock` to inform the user that the stock is low.

However, `low_stock` will compute a value every time the stock changes, which could lead to too many alarms until the order is performed and the situation is reverted.

To overcome this issue, we can define a stream `new_low` which only reports the changing points of `low_stock` (the let clauses allow to cleanly define local values):

```
ticks new_low := low_stock.ticks
define bool new_low t := let val  := low_stock(~t) in
                         let prev := low_stock(<t) in
                         if val != prev then val else notick
```

Finally, the following specifications allows emitting an alarm (of type *Unit*) every time the stock is low for a certain length of time `length`:

```
const length := 2h

ticks alarm := new_low.ticks
define T alarm t := if !new_low(~t)
                    then infty
                    else length

ticks long_low := delay alarm
define unit long_low t := ()
```

We can define a stream variable to report when there is a long period of low stock as defined by *long_low*, but also to report if after a certain amount of time `report_time` the stock is ok using the following specification:

```
const report_time := 8h

ticks clock_reset := report.ticks U {0}
define T clock_reset t := report_time

ticks report := delay clock_reset U long_low.ticks
define bool report t := isticking(long_low)
```

The dependency graph of the whole specification is shown in Figure 3.5. Since the dependency

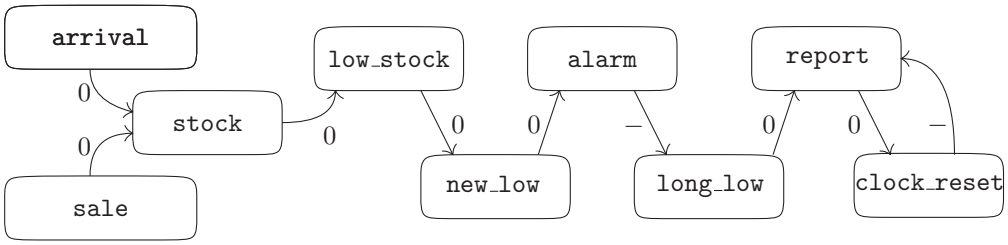


Figure 3.5: Dependency graph of the specification in Example 5

graph does not have a cycle with only positive arcs, the specification is well-formed and consequently well-defined.

3.6 Operational semantics

The semantics of *Striver* specifications introduced in the previous section is denotational in the sense that this semantics associates for a given input stream valuation exactly one output stream valuation, but does not provide a procedure to compute the output streams, let alone do it incrementally.

We provide in this section an operational semantics that computes the output incrementally.

We fix a specification φ with dependency graph G and we let $G^{=}$ be its pruned dependency graph (obtained from G by removing $\xrightarrow{0}$ edges). We also fix $<$ to be an arbitrary total order between stream variables that is a reverse topological order of $G^{=}$.

We first present an online monitoring algorithm that stores the full history computed so far for every output stream variable. Later we will provide bounds on the portion of the history that needs to be remembered by the monitor, showing that only a bounded number of events needs to be recorded, and that this bound depends only on the size of the specification and not on the length of trace. This modified algorithm is a trace-length independent monitor for efficiently monitorable *Striver* specifications.

The algorithm maintains the following state (H, t_q):

- **History**: H is a finite event stream one for each output stream variable. We use H_y for the event stream prefix for stream variable y.
- **Quiescence time**: t_q is the time up to which all output streams have been computed.

The monitor runs a main loop, calculating first the next relevant time t_q for the monitoring evaluation and then computing all outputs (if any) for time t_q. We show that no event exists in any stream in the interval between two consecutive quiescence time instants. We assume that at time t, the next event for every input stream is available to the monitor, even though knowing that there is no event up-to some t_q is sufficient.

The core observation follows from Lemma 1, which limits the information that is necessary to compute whether stream y at instant t contains an event (t, d). All this information ins contained

Algorithm 1 MONITOR: Online Monitor

```
1:  procedure MONITOR
2:      H_s ← ⟨⟩ for every s
3:      t_q ← −∞
4:      loop                                                          ▷ Step
5:          t_q ← min_{s∈O} {t | t = VOTE(H, T_s, t_q)}
6:          if t_q = ∞ then break
7:          for s in G^= following < do
8:              if t_q ∈ ⟦T_s⟧_H then
9:                  v ← ⟦V_s⟧_H(t_q)
10:                 if v ≠ ⊥^𝔻_notick then
11:                     H_s ← H_s ++ (t_q, v)                         ▷ Updates history H
12:                     emit(t_q, v, s)
13:         end for
14:     end loop
```

Algorithm 2 VOTE: Compute the next ticking instant

```
15: function VOTE(H, expr, t)
16:     switch expr do
17:         case delay(s)
18:             if (t′ + v) > t (where (t′, v) = last(H_s)) then return t′ + v
19:             else return ∞
20:         case {c}
21:             if c > t then return c
22:             else return ∞
23:         case a ∪ b
24:             return min(VOTE(H, a, t), VOTE(H, b, t))
25:         case y.ticks with y ∈ O
26:             return VOTE(H, T_y, t)
27:         case s.ticks with s ∈ I
28:             return succ_>(σ_s, t_q)
```

in H, so we write $[\![T_y]\!]_H$ and $[\![V_y]\!]_H$ to remark that only H is needed to compute $[\![T_y]\!]_\sigma$ and $[\![V_y]\!]_\sigma$.

The main algorithm, MONITOR, is shown in Algorithm 1. Lines 2 and 3 set the initial quiescence time and history. The main loop continues until no more events can be generated. Line 5 computes the next quiescence time, by taking the minimum instant after t_q^{prev} at which some output stream may tick. A stream y "votes" (see Algorithm 2) for the next possible instant at which its ticking equation T_y might contain a value. Consequently, if no input stream votes for an earlier time it is guaranteed that no ticking equation will contain a value t lower than the lowest vote. Note that recursive calls at line 28 terminate because the graph $G^=$ is acyclic (that is, the specification is well-formed).

The algorithm is following a topological order over the $G^=$, so the past information in Lemma 1 is contained in H. The following result shows that, assuming that σ_I is non-zeno, the output is also non-zeno. Hence, for every instant t, the algorithm eventually reaches $t_q > t$ in a finite number of

executions of the main loop.

Lemma 2. MONITOR generates non-zeno output for a given non-zeno input.

Proof. Note that events are generated in strictly increasing time for every stream, because the quiescent time t_q decided in line 5 is greater than the current time. However, that does not imply non-zenoness because some time domains (like the reals and the rationals) accept infinite sequences of increasing time stamps do not pass a given instant t.

Now, we first show that if the output generated by the monitor is zeno for time t (that is, there is no bound on the executions of the loop body that make $t_q > t$) then the execution is also zeno for time $t - \epsilon$. The lemma then follows because, by repeating the result $\frac{t}{\epsilon}$ times we will obtain that there is a zeno execution that does not pass $t - \epsilon\frac{t}{\epsilon} = 0$, but the second execution already passes 0.

Consider one such offending t. There must be an output stream variable x that votes infinitely many times in the infinite sequence of increasing quiescence times that never pass t. Let x be the lowest such stream variable in $(G^{=}, <)$. Consider the ticking expression for x. Since $\cup$ collects the votes for its sub-expressions, it follows that some sub-expression votes for infinitely many quiescent times in the sequence. The sub-expression cannot be $s.ticks$, because s would be lower than x in $<$ (contracting that x is minimal). Hence, the sub-expression voting infinitely many times is of the form **delay**(s). Then, all these votes are caused by different events in H_s that are ticks of s that happened earlier than $t - \epsilon$. □

We finally show that the output of MONITOR is an evaluation model. We use $H_s^i(\sigma_I)$ for the history of events H_s after the i-th execution of the loop body, and $H_s^*(\sigma_I)$ for the sequence of events generated after a continuous execution of the monitor. Note that $H_s^*(\sigma_I)$ can be a finite sequence of events (if the input is bounded and no repetition is introduced in the specification using **delay**) or an infinite sequence of events. In the first case, the vote is eventually ∞ and the monitoring algorithm halts.

Theorem 2. Let σ_I be an input event stream, and let σ_O consist of $\sigma_x = H_x^*(\sigma_I)$ for every output stream x. Then (σ_I, σ_O) is an evaluation model of φ.

Proof. Let σ be (σ_I, σ_O). By Lemma 2 the sequence of quiescent times is a non-zeno sequence. We show by induction on the votes of MONITOR that for every quiescent time t_q, σ is an evaluation model up-to t_q, that is $H_x^*|_{t_q} = [\![T_x, V_x]\!]_\sigma|_{t_q}$.

Consider a quiescent time t_q^{prev} and let $t_y = \text{VOTE}(H, y.ticks, t_q^{\text{prev}})$. We first show that for every output stream y, $t_y \in [\![T_y]\!]_\sigma$ and for no t' with $t_q^{\text{prev}} < t' < t_y$, $t' \in [\![T_y]\!]_\sigma$. This results follows by induction on $<$, by Lemma 1 which guarantees that only the past is necessary to evaluate $[\![T_y]\!]_\sigma$, and by our assumption that σ is an evaluation model up-to t_q^{prev}. Now, let t_q be the next quiescence time after t_q^{prev} chosen in line 5. We show, again by induction on $<$, that for every output stream variable y, H_y contains an event (t_q, v) if and only if $t_q \in [\![T_y]\!]_\sigma$ (which we showed above), and $v = [\![V_y]\!]_\sigma = [\![V_y]\!]_H$ as computed in line 9. Hence, all events in H_y satisfy that $(t_q, v) \in [\![T_y, V_y]\!]_\sigma$ and all events $(t_q, v) \in [\![T_y, V_y]\!]_\sigma$ are added to H_y at quiescence time t_q. Since only quiescent times can

satisfy $[\![T_y]\!]_\sigma$, it follows that σ is an evaluation model up-to t_q if σ is an evaluation model up-to t_q^{prev}, as desired. Finally, since the set of quiescent times is non-zeno, for every t there is a finite number n of executions of loop body after which $t_q^n \geq t$. Then, after n rounds σ is guaranteed to be an evaluation model up-to t. Since t is arbitrary, it follows that σ is an evaluation model. □

Corollary 1. Let φ be a well-formed specification, σ_I a non-zeno input stream and H^* the result of MONITOR. Then, H^* is the only evaluation model for input σ_I, and H^* is non-zeno.

Trace Length Independent Monitoring

The algorithm MONITOR shown above computes incrementally the only possible evaluation model for a given input stream, but this algorithm stores the whole prefix H_y for every output stream variable y. We show now a modification of the algorithm that is trace length independent, based on flattening the specification.

A specification is *flat* if every occurrence of an offset expression in every T_y is either $x(<\tilde{}\ \mathbf{t})$ or $x(\ll \mathbf{t})$. In other words, there can be no nested term of the form $x(<\tilde{}\ (y<\tilde{}\ \mathbf{t}))$ or $x(<\tilde{}\ (y\ll \mathbf{t}))$ or $x(\ll (y<\tilde{}\ \mathbf{t}))$ or $x(\ll (y\ll \mathbf{t}))$.

We first show that every specification can be transformed into a flat specification. The flattening applies incrementally the following steps to every nested term $x(E(y\ll \mathbf{t}))$, where E is an arbitrary offset term:

1. introduce a fresh stream s with equations $T_s = y.ticks$ and $V_s = x(E(\mathbf{t}))$
2. replace every occurrence of $x(E(y \ll \mathbf{t}))$ by $s(<\mathbf{t})$.

Example 6. Consider the following specification of a continuous integration process in software engineering. The intended meaning is to report in `faulty` those commits to a repository that fail the unit tests.

```
input commit_id commits, unit push, bool tests
ticks faulty                := tests.ticks
define commit_id faulty t   := if tests(~t)
                               then notick
                               else commits(<push<<t)
```

After applying the flattening process the specification becomes:

```
define commit_id faulty t := if tests(~t) then notick else s(<t)
                     ticks s := push.ticks
define commit_id s t      := commits(<t)
```

Here, `s` stores the `commit_id` of the last commit at the point of a `push`, which is precisely the information to report at the time of a `faulty` commit. □

Lemma 3. Let φ be a specification. There is an equivalent flat specification φ' that is linear in the size of φ.

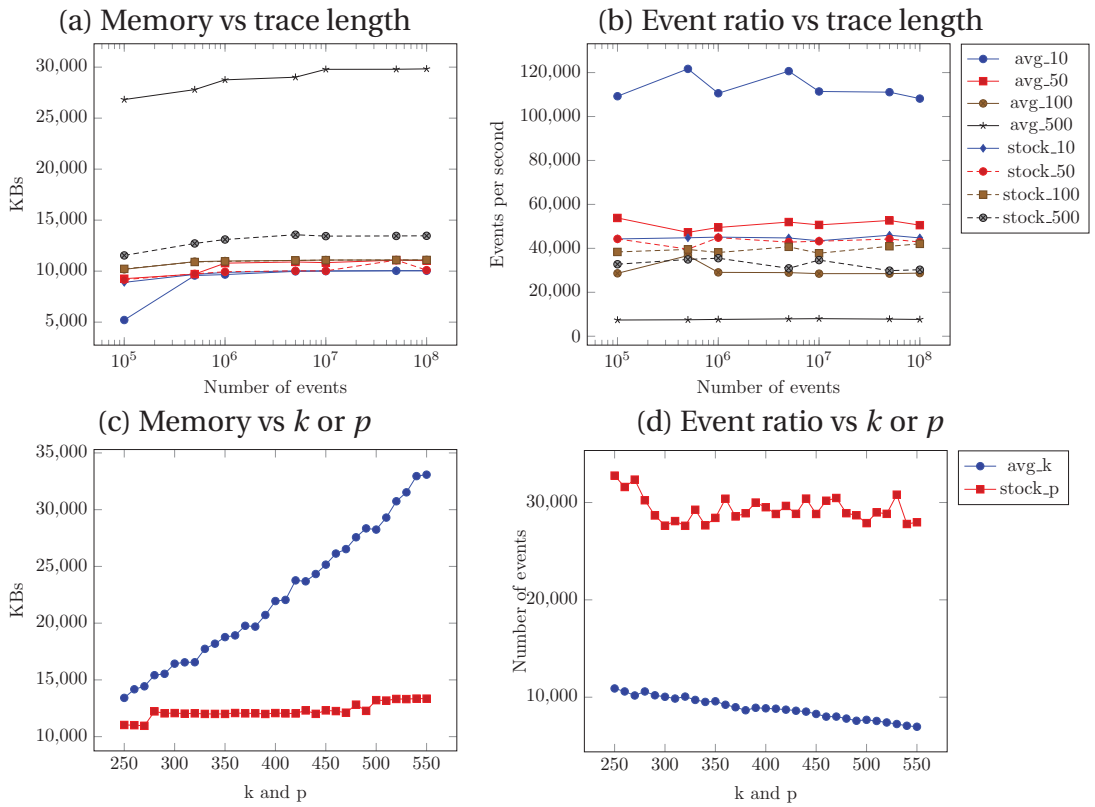


Figure 3.6: Empirical evaluation

Now, let φ' be the flat specification obtained from φ and let y be an output stream variable. Consider the cases for offset sub-expressions in the computation of $[\![V_y]\!]_H(t)$ in line 9 of MONITOR:

- $s <\tilde{} \ t$: the evaluation fetches the value H_s at time t (if s ticks at t) of at the previous ticking time (if s does not tick at t).

- $s \ll t$: the evaluation fetches the value H_s at the previous ticking time of s.

In either case, only the last two elements of H_s are needed. The similar argument can be made to compute T_y because only the last event of s is needed for **delay**(s).

Hence, to evaluate MONITOR on flat specifications, the algorithm only needs to maintain the last two elements in the history for every output stream variable to compute the next value of every value and ticking equation.

Theorem 3. Every flat specification φ can be monitored online with linear memory in the size of the specification and independently of the length of the trace. Moreover, every step can be computed in linear time on the size φ.

Proof. We apply the flattening step until every output stream definition is flat. □

3.7 Empirical evaluation

We report an empirical evaluation of a prototype *Striver* implementation, written in the Go programming language[2]. We measure the memory usage and time per event for two collections of specifications.

The first collection, from Example 3, computes the stocks of p independent products:

```
input int sale_1
input int arrival_1
...
input int sale_p
input int arrival_p

ticks stock_1 := sale_1.ticks U arrival_1.ticks
define int stock_1 t := stock_1(<t,0) +
    (if isticking(arrival_1) then arrival_1(~t) else 0) -
    (if isticking(sale_1   ) then sale_1(~t   ) else 0)
...
ticks stock_p := sale_p.ticks U arrival_p.ticks
define int stock_p t := stock_p(<t,0) +
    (if isticking(arrival_p) then arrival_p(~t) else 0) -
    (if isticking(sale_p   ) then sale_p(~t   ) else 0)
```

These specifications contain a number of streams proportional to p, where each defining equation is of constant size.

The second collection computes the average of the last k sales of a fixed product, via streams that tick at the selling instants and compute the sum of the last k sales:

```
input int sale

ticks denom := sale.ticks
define int denom t := if denom(<t) == k
                      then k
                      else denom(<t,0)+1

ticks sumlastk := sale.ticks
define int sumlastk t := sumlastk(<t,0) +
                         sale(~t) -
                         sale(<sale<<sale<<...<<t, 0)

ticks avgk := sale.ticks
```

[2] *Striver* is available at `http://github.com/imdea-software/striver`

```
define int avgk t := sumlastk / denom
```

The resulting specifications have depth proportional to k.

We instantiate k and p from 10 to 500 and run each resulting specification with a set of generated input traces. We run the experiments on a virtual machine on top of an Intel Xeon at 3GHz with 32GB of RAM, and measure the average memory usage (using the OS) and the number of events processed per second.

In a first experiment, we run the synthesized monitors with traces of varying length (top two plots in Figure 3.6). The results illustrate that the memory needed to monitor each specification is independent of the length of the trace (the curves are roughly constant). Also, the ratio of events processed is independent of the length of the trace.

In the second experiment, we fix a trace of 1 million events and run the specifications with k and p ranging from 250 to 550. The results (lower diagrams) indicate that the memory needed to monitor `stock_p` is independent of the number of products while the memory needed to monitor each `avg_k` specification grows linearly with k. Recall that theoretically all specifications can be monitored with memory linearly on the size of the specification.

3.8 Comparison with *TeSSLa*

TeSSLa [35] is a temporal stream-based specification language for Stream Runtime Verification which allows recursion and offers a collection of building blocks which can be composed to build a specification.

The building blocks to manipulate and create streams of asynchronous events are members of e, defined recursively in the following way:

$$e ::= \mathbf{nil} \mid \mathbf{unit} \mid x \mid \mathbf{lift}(f)(e,\ldots,e) \mid \mathbf{time}(e) \mid \mathbf{last}(e,e) \mid \mathbf{delay}(e,e)$$

and the semantics of each building block are the following:

- **nil** represents the empty stream, this is, the stream without events.
- **unit** represents the stream with a single unit event at timestamp zero.
- x is the name of an input or output stream.
- **lift**$(f)(e,\ldots,e)$ is the stream resulting from applying function f to the values of its argument streams interpreted as piece-wise constant signals; once all of them have been initialized.
- **time**(e) represents the stream of the timestamps of another stream, reported every time this signal generates a value.
- **last**(e,e) represents the stream resulting from returning the previous value of the first stream at the timestamps of the second.

- **delay**(e, e) The delay operation takes delays as its first argument. After a delay has passed, a unit event is emitted. A delay can only be set if a reset event is received via the second argument, or if an event is emitted on the output.

The equivalent *Striver* specifications of *TeSSLa* operators are defined as follows:

- **nil:** the stream $x = nil$ is defined as:

```
ticks x := {0}
define void x t := notick
```

- **unit:** the stream $x = unit$ is defined as:

```
ticks x := {0}
define unit x t := ()
```

- **lift**: the stream $x = lift\langle f, s_0, \ldots, s_n \rangle$:

```
input A0 s0
...
input An sn

ticks x := s0.ticks U ... U sn.ticks
define B x t :=
    if (s0<~t == outside || ... || sn<~t == outside)
    then notick
    else f(s0(~t),...,sn(~t))
```

- **time:** the stream $x = time\langle s \rangle$ is defined as:

```
input A s

ticks x := s.ticks
define Time x t := t
```

- **last**: the stream $x = last\langle s_0, s_1 \rangle$:

```
input A0 s0
input A1 s1

ticks x := s1.ticks
define A0 x t := s0(<t)
```

- **delay:** The stream $x = delay\langle s_0, s_1\rangle$ is defined as:

```
input Time_eps s0
input A s1
ticks x_aux := s0.ticks U s1.ticks
define Time_eps x_aux t :=
    if isticking(s1)
    then infty
    else
      if x_aux(<t,infty) = infty || x_aux(<t) + x_aux<<t <= t
      then s0(~t)
      else notick
ticks x := delay x_aux
define unit x t := ()
```

Example 7. We consider the specification that allows counting events in a given input stream, which is a built-in block in *TeSSLa* (or a recursive definition in *TeSSLa* 2.0). One equivalent *Striver* specification is the following:

```
input A s

ticks event_count := s.ticks U {0}
define int x t := if isticking(s)
                  then event_count(<t,0) + 1
                  else 0
```

3.9 Real use case

ElasTest is a platform whose goal is to ease end-to-end testing of distributed systems [55]. *ElasTest* aims to provide an easy deployment process and easy access to the necessary services usually involved in an end-to-end test, along with easy-to-use tools to show and analyze logs and metrics of all elements involved in an end-to-end test.

The goal of this public funded project is to design and implement a tool for helping developers to test and validate complex distributed systems.

The *ElasTest* platform is based on three principles:

- *Test orchestration*: Combining intelligently testing units for creating a more complete test suite following the "divide and conquer" principle.
- *Instrumentation and monitoring*: Customizing the System Under Test (SuT) infrastructure so that it reproduces real-world operational behavior and allowing gathering relevant information during testing.

- *Test recommendations*: Using machine learning and cognitive computing for recommending testing actions and providing testers with friendly interactive facilities for decision taking.

The *ElasTest* platform itself is designed following a microservices architecture.

One of these microservices, related to the second principle, is the *ElasTest Monitoring Service* (EMS), the component in charge of providing a monitoring infrastructure suitable for inspecting executions of a SuT and the *ElasTest* platform itself in real time.

This component allows the user and the platform to deploy *Monitoring Machines* able to process events in real time and generate complex, higher level events from the incoming stream of events. Such functionality can help to better understand what's happening in the execution of the test, detect anomalies, correlate issues, and even stress the tests automatically; all of which aims to maximize the chances of uncover bugs and their causes.

These Monitoring Machines are compiled into *Striver* specifications, which are executed using the Go *Striver* engine mentioned in Section 3.7.

The trace-length resource independence proven in Section 3.6 and witnessed in the empirical evaluation results ensures that the Monitoring Machines deployed to the ElasTest Monitoring Service can digest a high amount of events and run for long periods of time without experiencing a detriment on performance.

The code and the documentation of the ElasTest Monitoring Service are available at `https://github.com/elastest/elastest-monitoring-service/`.

Chapter 4

Extensions of Striver

We now present three extensions to the basic *Striver* introduced previously.

Accessing successors.

The first extension allows accessing future events, via the dual of the offset operators $x >\tilde{}\ e$ and $x \gg e$, which are added to the syntax of the *offset expressions*:

$$\tau_x := x <\tilde{}\ \tau' \mid x \ll \tau' \mid x >\tilde{}\ \tau' \mid x \gg \tau'$$
$$\tau' := \mathbf{t} \mid \tau_z \text{ for } z \in Z$$

We also define the corresponding syntactic sugar to access the successor value $x(e>)$, $x(e\tilde{})$, $x(e,d>)$ and $x(e,d\tilde{})$.

The type $\mathbb{T}_\epsilon$ of stream w in constructor **delay** w is defined as $\mathbb{T}_\epsilon = \{t \mid t \geq \epsilon\}$ if $\epsilon > 0$ and as $\mathbb{T}_\epsilon = \{t \mid t \leq \epsilon\}$ if $\epsilon < 0$. A **delay** w with a positive ϵ in the ticking expression of a defined stream x indicates an influence from the past $w \xrightarrow{-} x$. If ϵ is negative, then **delay** w indicates an influence from the future $w \xrightarrow{+} x$.

As for *LOLA*, well-formedness can be guaranteed as long as all strongly connected components in the dependency graph contain only $\xrightarrow{-}$ and $\xrightarrow{0}$ edges, or only $\xrightarrow{+}$ and $\xrightarrow{0}$ edges, and additionally, there is no cycle with only $\xrightarrow{0}$ edges. This guarantees that there is no cyclic dependency, as every stream either depends on itself in the future or in the past (or none at all).

Using successors, we can define a stream to detect if a file is opened and closed with no data read or written into it using the following specification:

Example 8. Consider the type `FILE_EVENT` to be `{OPEN, WRITE, READ, CLOSE}`. We can define the signal `nodataaccess` to emit a signal of type *Unit* if a file is opened and closed but its data is not accessed.

```
input FILE_EVENT file_events

ticks nodataaccess := file_events.ticks
```

```
define unit nodataaccess t := if file_events(~t) == OPEN &&
                                 file_events(t>) == CLOSE
                              then ()
                              else notick
```

All Delays.

This extension allows defining tick sets that consider all delays.

The ticking expressions are extended with an operator **delayall**:

$$\alpha := \{c\} \mid v.\textbf{ticks} \mid \alpha \cup \alpha \mid \textbf{delayall}\ w \mid \textbf{delay}\ w$$

with the following semantics:

$$[\![\textbf{delayall}(w)]\!]_\sigma \stackrel{\text{def}}{=} \{t' \mid \text{ there is a } t \in dom(\sigma_w) \text{ such that } t + \sigma_w(t) = t'\}$$

This extension requires only to change VOTE to accommodate for a set of possible pending delays and not just a single delay.

Intuitively, the **delayall** operator is similar to **delay**, with the difference that it does not get reset when a new event is received before the alarm set previously times out. Instead, the new alarm is added to an internal queue and all of them generate a tick at the corresponding times.

In general, this cannot be implemented in finite memory for arbitrary event rates and delays, but MONITOR works for online monitoring this construct.

The new **delayall** operator is useful to define windows, as described in the next extension.

Windows.

The last extension allows implementing computations over precise windows, like "*count the number of events in every window of one second*".

This cannot be described in *TeSSLa* [35], which is limited to finite memory monitors, or in RTLola [36] because this specification is not isochronous.

Note that this property cannot be monitored by splitting the time in intervals of one second and counting the events in each of the intervals obtained (as in RTLola) as this approach misses the case of counting the events in part of one window and the remaining time in the adjacent window.

The main idea of this extension is to enrich time with a tag, in such a way that every tick carries an additional value (we called this extension *dependent time*).

Then, **delay** and **delayall** are enriched with the ability to use tagged time streams, with the caveat that the **U** combinator must now indicate how to combine tags.

Consider the following example:

```
input int s

ticks ones_at_s := s
define int ones_at_s t tag := 1

ticks remove_s := s
define (int, Time) remove_s t tag := (-1,5)

ticks wcount :=  ones_at_s U delayall remove_s
define int wcount t tag := wcount(<t,0) + tag
```

The output stream `ones_at_s` constructs a stream that ticks as s and has a constant 1 as its only value, while the output stream `remove_s` is defined to create a tick with tag -1 and 5 seconds as the time for **delayall**.

The stream `wcount` must only be computed when a new event arrives in s (adding 1) or when an event leaves the window (substracting 1), which is monitored with a constant number of operations per event, but requires storing a number of events that depends on the event rate.

A specification that aggregates the value of all events within a window of 5 seconds can be defined similarly.

Example 9. The following specification adds the value of the event stream s within a sliding window of 5 seconds:

```
input int s

ticks remove_s := s
define (int, Time) remove_s t tag := (s(~t),5)

ticks  int waggr :=  s  U delayall remove_s
define int waggr t tag := waggr(<t,0) + aux
```

Chapter 5

Conclusions and future work

We have introduced *Striver*, an efficiently monitorable specification language with explicit time and offset reference for the Stream Runtime Verification of timed event streams. We have presented a trace-length independent online monitoring algorithm for *Striver*, extensions of the language and compared the expressive power against related formalisms. We have also shown a real use case of *Striver* for the monitorization of cloud applications.

Future work includes the extension of the language with parametrization, (like in QEA, FOMTL and Lola2.0 [56]), to dynamically instantiate monitors for observed data items. We are also studying offline evaluation algorithms, and algorithms that tolerate deviations in the time-stamps of input events and asynchronous arrival of events from the different input streams.

Bibliography

[1] Compaq meeting minutes. https://s3.amazonaws.com/files.technologyreview.com/p/pub/legacy/compaq_cst_1996_0.pdf, 1996. Accessed: 2018-07-16.

[2] Who Coined 'Cloud Computing'? https://www.technologyreview.com/s/425970/who-coined-cloud-computing/. Accessed: 2018-07-16.

[3] Announcing Amazon Elastic Compute Cloud. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/. Accessed: 2018-07-16.

[4] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing, 2011.

[5] Chris Richardson. Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html. Accessed: 2018-07-16.

[6] Anil Karmel, Ramaswamy Chandramouli, and Michaela Iorga. NIST Definition of Microservices, Application Containers and System Virtual Machines, 2016.

[7] What is Cloud Computing? https://aws.amazon.com/what-is-cloud-computing/. Accessed: 2018-07-16.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. https://www.w3.org/TR/soap, 2000.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3c note, World Wide Web Consortium, March 2001.

[10] The OpenAPI Specification. https://github.com/OAI/OpenAPI-Specification. Accessed: 2018-07-16.

[11] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.

[12] Nancy Leveson. Medical Devices: The Therac-25, 1985.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[14] S. Halle and R. Villemaire. Runtime Enforcement of Web Service Message Contracts with Data. *IEEE Transactions on Services Computing*, 5(2):192–206, April 2012.

[15] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification, 2008.

[16] Klaus Havelund and Allen Goldberg. Verify your runs. In *Proc. of VSTTE'05*, LNCS 4171, pages 374–383. Springer, 2005.

[17] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *J. Logic Algebr. Progr.*, 78(5):293–303, 2009.

[18] *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.

[19] Klaus Havelund and Grigore Roşu. Synthesizing Monitors for Safety Properties. In *Proc. of TACAS'02*, LNCS 2280, pages 342–356. Springer, 2002.

[20] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Proc. of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, 2003.

[21] Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime Verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.

[22] Joël Ouaknine and James Worrell. Some Recent Results in Metric Temporal Logic. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, pages 1–13, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[23] Koushik Sen and Grigore Roşu. Generating Optimal Monitors for Extended Regular Expressions. *ENTCS*, 89(2):226–245, 2003.

[24] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[25] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In *Proc. of VMCAI'04*, LNCS 2937, pages 44–57. Springer, 2004.

[26] Klaus Havelund. Monitoring with Data Automata. In *ISoLA*, 2014.

[27] 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, *FM 2012: Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[28] Oded Maler and Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[29] Marcelo d'Amorim and Klaus Havelund. Event-based Runtime Verification of Java Programs. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM.

[30] Grigore Roşu and Klaus Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[31] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *Proc. of FM'06*, LNCS 4085, pages 573–586. Springer, 2006.

[32] 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 TIME'05*, pages 166–174. IEEE, 2005.

[33] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A Hard Real-Time Runtime Monitor. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 345–359, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams. In *Proc. of the 33rd Symposium on Applied Computing (SAC'18)*. ACM, 2018.

[35] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal Stream-based Specification Language. submitted.

[36] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time Stream-based Monitoring. *CoRR*, abs/1711.03829, 2017.

[37] Defenses Against TCP SYN Flooding Attacks. `https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-34/syn-flooding-attacks.html`. Accessed: 2018-07-16.

[38] Sylvain Hallé and Simon Varvaressos. A Formalization of Complex Event Stream Processing, 09 2014.

[39] Sylvain Hallé and Raphael Khoury. Event Stream Processing with BeepBeep 3. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 81–88. EasyChair, 2017.

[40] 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, USA, 2006. ACM.

[41] Sylvain Hallé, Sébastien Gaboury, and Bruno Bouchard. Activity Recognition Through Complex Event Processing: First Findings. In *AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments*, 2016.

[42] C. Zang and Y. Fan. Complex event processing in enterprise information systems based on RFID. *Enterprise Information Systems*, 1(1):3–23, 2007.

[43] Alexandre Alves, Robin J. Smith, and Lloyd Williams. *Getting Started with Oracle Event Processing 11G*. Packt Publishing, 2013.

[44] `https://www.influxdata.com/time-series-platform/`. Accessed: 2018-07-16.

[45] `https://www.influxdata.com`. Accessed: 2018-07-16.

[46] `https://www.influxdata.com/time-series-platform/influxdb/`. Accessed: 2018-07-16.

[47] `https://www.elastic.co/elk-stack`. Accessed: 2018-07-16.

[48] `https://www.elastic.co`. Accessed: 2018-07-16.

[49] `https://www.elastic.co/products/elasticsearch`. Accessed: 2018-07-16.

[50] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems*, 3(3):263–286, Aug 2001.

[51] Eamonn Keogh and Padhraic Smyth. A Probabilistic Approach to Fast Pattern Matching in Time Series Databases. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, KDD'97, pages 24–30. AAAI Press, 1997.

[52] Felipe Gorostiaga and César Sánchez. Striver: A Stream-Based Specification Language for Real-Time Signals. 2018. Paper under submission.

[53] Flaviu Cristian and Christof Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[54] Martin Leucker. Teaching Runtime Verification. In *Proc. of RV'11*, number 7186 in LNCS, pages 34–48. Springer, 2011.

[55] `https://elastest.io`. Accessed: 2018-07-16.

[56] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A Stream-Based Specification Language for Network Monitoring. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.