Noname manuscript No.
(will be inserted by the editor)

A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software)

César Sánchez · Gerardo Schneider · Wolfgang Ahrendt · Ezio Bartocci · Domenico Bianculli · Christian Colombo · Yliés Falcone · Adrian Francalanza · Srđan Krstić · João M. Lourenço · Dejan Nickovic · Gordon J. Pace · Jose Rufino · Julien Signoles · Dmitriy Traytel · Alexander Weiss

the date of receipt and acceptance should be inserted later

Corresponding authors: César Sánchez E-mail: <cesar.sanchez@imdea.org> and Gerardo Schneider E-mail: <gersch@chalmers.se>.

C. Sánchez
IMDEA Software Institute, Spain

G. Schneider
University of Gothenburg, Sweden

W. Ahrendt
Chalmers University of Technology, Sweden

E. Bartocci
TU Wien, Austria

D. Bianculli
University of Luxembourg, Luxembourg

C. Colombo · A. Francalanza · G. Pace
University of Malta, Malta

Y. Falcone
Univ. Grenoble Alpes, CNRS, Inria, LIG, France

S. Krstić · D. Traytel
ETH Zürich, Switzerland

J. Lourenço
Universidade Nova de Lisboa, Portugal

D. Nickovic
Austrian Institute of Technology, Austria

J. Rufino
Universidade de Lisboa, Portugal

J. Signoles
CEA LIST, Software Reliability & Security Lab, France

A. Weiss
Accemic Technologies GmbH, Germany

Abstract Runtime verification is an area of formal methods that studies the dynamic analysis of execution traces against formal specifications. Typically, the two main activities in runtime verification efforts are the process of creating monitors from specifications, and the algorithms for the evaluation of traces against the generated monitors. Other activities involve the instrumentation of the system to generate the trace and the communication between the system under analysis and the monitor.

Most of the applications in runtime verification have been focused on the dynamic analysis of software, even though there are many more potential applications to other computational devices and target systems. In this paper we present a collection of challenges for runtime verification extracted from concrete application domains, focusing on the difficulties that must be overcome to tackle these specific challenges. The computational models that characterize these domains require to devise new techniques beyond the current state of the art in runtime verification.

1 Introduction

Runtime verification (RV) is a computing analysis paradigm based on observing executions of a system to check its expected behavior. The typical aspects of an RV application are the generation of a monitor from a specification and then the use of the monitor to analyze the dynamics of the system under study. RV has been used as a practical application of formal verification, and as a less *ad-hoc* approach complementing conventional testing and debugging. Compared to static formal verification, RV gains applicability by sacrificing completeness as not all traces are observed and typically only a prefix of a potentially infinite computation is processed. See [184,226] for surveys on RV, and the recent book [47].

Most of the practical motivations and applications of RV have been related to the analysis of software. However, there is a great potential for applicability of RV beyond software reliability if one generalizes to new domains beyond computer programs (like hardware, devices, cloud computing and even human centric systems). Novel applications of RV to these areas can have an enormous impact in terms of enabling new solutions or designs, and the potential increase in reliability in a cost effective manner. Many system failures through history have exposed the limitations of existing engineering methodologies and encouraged the study and development of novel formal methods. Ideally, one would like to validate a computational system prior to its execution. However, current static validation methods, such as model checking, suffer from practical limitations preventing their wide use in real large-scale applications. For instance, those techniques are often bound to the design stage of a system and suffer from the state-explosion problem (the unfeasibility to exhaustively explore all system states statically), or cannot handle many interesting behavioral properties. Thus, as of today many verification tasks can only realistically be undertaken by complementary dynamic analysis methods. RV is the discipline of formal dynamic analysis that studies how to detect and ensure, at execution time, that a system meets a desirable behavior.

Even though research on runtime verification has flourished in the last decade[1], a big part of the (European) community in the area has recently been gathered via a EU COST action initiative[2] in order to explore, among other things, potential areas of

[1] See the the conference series at `http://runtime-verification.org`.

[2] *Runtime Verification beyond Monitoring (ARVI)*: ICT COST Action IC1402 (`http://www.cost.eu/COST_Actions/ict/IC1402`)

application of RV, including finances, medical devices, legaltech, security and privacy, and embedded, cloud and distributed systems.

In this survey paper, we concentrate in the description of different challenging and exciting application domains for RV, others than programming languages. In particular we consider runtime verification in the following application domains:

Distributed systems: where the timing of observations may vary widely in a non-synchronised manner (Section 2).
Hybrid and embedded systems: where continuous and discrete behavior coexist and the resources of the monitor are constrained (Section 3).
Hardware: where the timing must be precise and the monitor must operate non disruptively (Section 4).
Security and privacy: where a suitable combination between static and dynamic analysis is needed (Section 5).
Transactional information systems: where the behavior of modern information systems is monitored, and the monitors must compromise between expressivity and non-intrusiveness (Section 6).
Contracts and policies: where the connection between the legal world and the technical is paramount (Section 7).
Huge, unreliable or approximated domains: where we consider systems that are not reliable, or aggregation or sampling is necessary due to large amounts of data (Section 8).

In all these cases, we first provide an overview of the domain, and describe sufficient background to present the context and scope. Then, we introduce the subareas of interest addressed in the section, and identify challenges and opportunities from the RV point of view. Sometimes the characteristics and applications are not specific to RV, and in these cases we prefer to describe them in their generality, with the intention to motivate their importance first, to later speculate on how RV can have an impact in these applications and what are the challenges to monitoring. Finally, we do not aim for completeness in the identification of the challenges and admittedly only identify a subset of the potential challenges to be addressed by the RV research community in the next years. We identify the challenges listed as some of the most important.

2 Distributed and Decentralized Runtime Verification

Distributed systems are generally defined as computational artifacts or components that run into execution units placed at different physical locations, and that exchange information to achieve a common goal. A localized unit of computation in such a setup is generally assigned its own process of control (possibly composed of multiple threads), but does not execute in isolation. Instead, the process interacts and exchanges information with other such remote units using the communication infrastructure imposed by the distributed architecture, such as a computer network [32,116,172].

Distributed systems are notoriously difficult to design, implement and reason about. Below, we list some of these difficulties.

– Multiple stakeholders impose their own requirements on the system and the components, which results in disparate specifications expressed in widely different formats and logics that often concern themselves with different layers of abstraction.

– Implementing distributed systems often involves collaboration across multiple development teams and the use of various technologies.
– The components of the distributed system may be more or less accessible to analysis, as they often evolve independently, may involve legacy systems, binaries, or even use remote proprietary services.
– The sheer size of distributed systems, their numerous possible execution interleaving, and unpredictability due to the inherent dynamic nature of the underlying architecture makes them hard to test and verify using traditional pre-deployment methods. Moreover, distributed computation is often characterized by a high degree of dynamicity where all of the components that comprise the system are not known at deployment (for example, the dynamic discovery of web services) — this dynamicity further complicates system analysis.

Runtime Verification (RV) is very promising to address these difficulties because it offers mechanisms for correctness analysis *after* a system is deployed, and can thus be used in a multi-pronged approach towards assessing system correctness. It is well-known that even after extensive analysis at static time, latent bugs often reveal themselves once the system is deployed. A better detection of such errors at runtime using dynamic techniques, particularly if the monitor can provide the runtime data that leads to the error, can aid system engineers to take remedial action when necessary. Dynamic analysis can also provide invaluable information for diagnosing and correcting the source of the error. Finally, runtime monitors can use runtime and diagnosis information to trigger reaction mechanisms correcting or mitigating the errors.

We discuss here challenges from the domain of *Distributed and Decentralized Runtime Verification* (DDRV), a broad area of research that studies runtime verification in connection with distributed or decentralized systems, or when the runtime verification process is decentralized. That it, this body of work includes the monitoring of distributed systems as well as the use of distributed systems for monitoring.

Solutions to some of these research efforts exist (see for instance [217,218,105,84,162,147,63,149,93]). We refer to [161] for a recent survey on this topic.

2.1 Context and Areas of Interest

In order to provide context to later describe the challenges for RV, we begin by describing some important characteristics of DDRV and then list some intended applications.

2.1.1 Characteristics

There are a number of characteristics that set DDRV apart from non-distributed RV. These characteristics also justify the claim that traditional RV solutions and approaches commonly do not necessarily (or readily) apply to DDRV. This, in turn, motivates the need for new mechanisms, theories, and techniques. Some characteristics were identified in [79,157,204], and recently revisited in [161].

Heterogeneity and Dynamicity. One of the reasons that makes distributed systems hard to design, implement and understand is that there are typically many participants involved. Each participant imposes its own requirements ending in a variety of specifications expressed in different formats. In turn, the implementation often involves the collaboration of multiple development teams using a variety of technologies. Additionally, the size and dynamic characteristics of the execution platform of distributed systems allow many possible interleavings of the behaviors of the participating components, which leads to an inherent unpredictability of the executions. Note that the existence of a set of interleavings and the necessity to explore or reason about alternative paths in this set is due to distributed systems being concurrent systems with asynchronous communication. The inherent dynamicity of distributed systems makes this set larger and more complex, and the exploration of the set harder.

Consequently, testing and verification with traditional pre-deployment methods are typically ineffective.

Distributed Clocks and Latency. Distributed systems can be classified according to the nature of the clocks: from (1) synchronous systems, where the computation proceeds in rounds, (2) timed asynchronous systems, where messages can take arbitrarily long but there is a synchronized global clock, (3) asynchronous distributed systems. In an asynchronous distributed system, nodes are loosely coupled, each having its own computational clock, due to the impracticality of keeping individual clock synchronized with one another. As a result of this asynchrony, the order of computational events occurring at distinct execution units may not be easy (or even possible) to discern.

Partial Failure. A requirement of any long-running distributed system is that, when execution units (occasionally) fail, the overall computation is able to withstand the failure. However, the independence of failure between the different components of a distributed system and the unavailability of accurately detecting remote failures, makes designing fail tolerant systems challenging. When designing a solution based on RV, the independence of failure between components is an important characteristic that must be handled by the monitoring infrastructure (for example re-synchronization between living components, rebooting monitors, etc).

Non-Determinism. Asynchrony implies fluctuations in latency, which creates unpredictability in the global execution of a distributed system. In addition, resource availability (*e.g.,* free memory) at individual execution units is hard to anticipate and guarantee. These sources of unpredictable asynchrony often induce nondeterministic behavior in distributed computations [154,155].

Multiple Administrative Domains and Multiple Accessibility. In a distributed system, computation often crosses administrative boundaries that restrict unfettered computation due to security and trust issues (*e.g.,* mistrusted code spawned or downloaded from a different administrative domain may be executed in a sandbox). Administrative boundaries also limit the migration and sharing of data across these boundaries for reasons of confidentiality. Also, different components may feature different accessibility when it comes to analysis, maintenance, monitorability, instrumentation, and enforcement. The connected technologies may range from proprietary to the public domain, from available source code to binaries only, from well-documented developments to sparsely documented (legacy) systems.

Mixed Criticality. The components and features of a distributed system may not be equally critical for the overall goal of the system. The consequences of malfunctioning of certain components are more severe than the malfunctioning of others. For instance, the failure of one client is less critical than the failure of a server which many clients connect to. Also, some components could be critical for preventing data or financial loss, or alike, whereas others may only affect performance or customer satisfaction.

Evolving Requirements. The execution of a distributed system is typically characterized by a series of long-running reactive computational entities (*e.g.,* a web server that should ideally never stop handling client requests). Such components are often recomposed into different configurations (for example, service-oriented architectures) where their intended users change. In such settings, it is reasonable to expect the correctness specifications and demands to change over the execution of the system, and to be composed of smaller specifications obtained from different users and views.

2.1.2 Applications

We briefly mention some of the existing or envisioned application areas of DDRV, namely concurrent software, new programming paradigms such as reversible computing [160], the verification of distributed algorithms or distributed data bases, privacy and security (intrusion detection systems, auditing of policies on system logs [60, 171], decentralized access control [307]), blockchain technology [248], monitoring software-defined networks with software defined monitoring, robotics (e.g., distributed swarms of autonomous robots), and home automation.

Enforcing Interleavings

Sometimes the system that one analyzes dynamically —using runtime verification— is distributed in nature. For example, multithreaded programs can suffer from concurrency errors, particularly when executing in modern hardware platforms, as multicore and multiprocessor architectures are very close to distributed systems. This makes the testing of concurrent programs notoriously difficult because it is very hard to explore the interleavings that lead to errors [31]. The work in [234] proposes to use enforcement exploiting user-specified properties to generate local monitors that can influence the executions. The goal is to improve testing by forcing promising schedules that can lead to violations, even though violations of the specified property can also be prevented by blocking individual threads whose execution may lead to a violation. The process for generating monitors described in [234] involves the decomposition of the property into local decentralized monitors for each of the threads.

Observing Distributed Computations

Checking general predicates in a distributed system is hard, since one has to consider all possible interleavings (which may be exponential in size). Techniques like *computation slices* [12, 31, 99, 242] have been invented as a datatype for the efficient distributed detection of predicates. Slices allow to circumvent an explicit exploration of a large set of interleaving paths by an implicit exploration of a smaller representation. Slices are a concise approximation of the computation, which are precise enough to detect the predicate because slices guarantee that if a predicate is present in a slice of a computation then the predicate occurred in some state of the computation.

Predicate detection can involve a long runtime and large memory overhead [99] except for properties with specific structure (that is, for some fragments of the language of predicates). Current efficient solutions only deal with sub-classes of safety properties like linear, relational, regular and co-regular, and stable properties. Even though most techniques for predicate detection ([12,109,242]) send all local events to a central process for inspection of its interleavings, some approaches (like [99]) consider purely distributed detection.

Monitor Decomposition and Coordination

Most approaches to monitoring distributed systems consider that the system is a black-box that emits events of interest, while others use a manual instrumentation and monitor placement. Some exceptions, for example [147,157,162,250,8,7], investigate how to exploit the hierarchical description of the system to generate monitors that are then composed back with the original system. The modified system shares the original decomposition (of course implementing its functionality) and includes the monitors embedded, but this approach requires to have access to the system description and is specific to a given development language. Although the work in [147] does not specifically target distributed systems, the compiler can generate a distributed system in which case the monitor will be distributed as well. A similar approach is presented in [30,92,93,162], where a framework for monitoring asynchronous component-based systems is presented based on actors—self contained software entities that are easily distributed.

Monitoring Efficiency

Most RV works assume a single monitor that receives all events and calculates the verdicts. Even though a single monitor can be implemented for decentralized and distributed systems by sending all information to a central monitor, distribution itself can be exploited to coordinate the monitoring task more efficiently. Many research efforts study how to gain more efficient solutions by exploiting the locality in the observations to also perform partially the monitoring task locally as much as possible. For example, the approaches in [30,92,147,93,31] exploit the hierarchical structure of the system to generate local monitors, and [94,162,5] exploit the structure and semantics of the specification. In [7], the authors show how decentralized monitor specifications can be consolidated into regular descriptions that guarantee bounded state space. Lowering overheads is also pursued in [105] by offloading part of the monitoring computation to the computing resources of another machine.

When atomic observations of the monitored system occur locally, monitors can be organized hierarchically according to the structure of the original specification [65,66, 157,104]. Substantial savings in communication overheads are obtained because often a verdict is already reached in a sub-formula. All these results are limited to LTL and regular languages in [144]. Decentralized monitoring assumes that the computation proceeds in rounds, so distributed observations are synchronized and messages eventually arrive. The assumption of bounded message delivery is relaxed in [104].

Fault Tolerance

One of the main and most difficult characteristics of distributed systems is that failures can happen independently (see [159]). Most of the RV efforts that consider distributed systems assume that there are no errors, that is, nodes do not crash and messages are not corrupted, lost, duplicated or reordered. Even worse, failure dependencies between components can be intricate and the resulting patterns of behaviors can be difficult to predict and explain. At the same time, one of the common techniques for fault tolerance is the replication of components so this is a promising approach for monitoring too [158]. For example, [153] studies the problem of distributed monitoring with crash failures, where events can be observed from more than one monitor, and where the distributed monitoring algorithm tries to reach a verdict among the surviving monitors.

Another source of failure is network errors, studied in [54,63,4], which targets the incomplete knowledge caused by network failures and message corruptions and attempts to handle the resulting disagreements. Node crashes are handled because message losses can simulate node crashes by ignoring all messages from the crashed node.

2.2 Challenges

The characteristics outlined bring added challenges to obtain effective DDRV setups.
C 2.1. Distributed Specifications. It is a well-established fact that certain specifications cannot be adequately verified at runtime [5,98,146,156,272,4,7]. The partial ordering on certain distributed events, due to *distributed clocks* hinders the monitoring of temporal specifications requiring a specific relative ordering of these events [31]. As such, the lack of a global system view means that even fewer specifications can be monitored at runtime. Even though some work exists proposing specific languages tailored to distributed systems [298], the quest for expressive and tractable languages is an important and challenging goal.

C 2.2. Monitor Decomposition, Placement, and Control. The runtime analysis carried out by monitors needs to be distributed and managed across multiple execution nodes. As argued originally in [157], and later investigated empirically in works such as [31,66], the decomposition and placement of monitoring analysis is an important engineering decision that affects substantially the overheads incurred such as the number and size of messages, the communication delay, the spread of computation across monitors [136]. Such placement also affects the administrative domains under which event data is analyzed and may compromise confidentiality restrictions and lead to security violations that may be due to the communication needed by monitors to reach a verdict (for instance if monitors communicate partial observations or partial evaluations of the monitored properties).

C 2.3. Restricted Observability. The flip side of security and confidentiality constraints in distributed systems translates into additional observability constraints that further limit what specifications can be monitored in practice. Distributed monitors may need to contend with traces whose event data may be obfuscated or removed in order to preserve confidentiality which, in turn, affects the nature of the verdicts that may be given [4,179].

C 2.4. Fault Tolerance. DDRV has to contend with the eventuality of failure in a distributed system [63]. Using techniques involving replication and dynamic reconfiguration of monitors, DDRV can be made tolerant to *partial failure*. More interestingly, fault-tolerant monitoring algorithms could provide reliability to the monitors. A theory allowing to determine which specifications combined with which monitoring algorithms could determine the guarantees that should be investigated.

C 2.5. Deterministic Analysis. Since monitoring needs to be carried out over a distributed architecture, this will inherently induce non-deterministic computation. In spite of this, the monitoring analysis and the verdicts reported need to feature aspects such as strong eventual consistency [136] or observational verdict determinism [155, 154], and conceal any internal non-determinism. In practice, this may be hard to attain (*e.g.,* standard determinization techniques on monitors incur triple exponential blowup [6]); non-deterministic monitor behavior could also compromise the correctness of RV setup and the validity of the verdicts reported [154].

C 2.6. Limits of Monitorability. Distributed systems impose further limitations on the class of properties that can be detected (see [314,272,67,145,146,120,156,5,8] for notions of monitorability for non-distributed systems and [31,136] for decentralized systems [137]). Associated with the challenge of exploring new specification languages for monitoring distributed systems, there is the need to discern the limitations of what can be detected dynamically.

3 Hybrid Systems

Hybrid systems (HS) [188] are a powerful formal framework to model and to reason about systems exhibiting a sequence of piecewise continuous behaviors interleaved with discrete jumps. In particular, *hybrid automata* (HA) extend finite state-based machines with continuous dynamics (generally represented as ordinary differential equations) in each state (also called *mode*). HS are suitable modelling techniques to analyze safety requirements of *Cyber-Physical Systems* (CPS). CPS consist of computational and physical components that are tightly integrated. Examples include engineered (i.e., self-driving cars), physical and biological systems [51] that are monitored and/or controlled through sensors and actuators by a computational embedded core. The behavior of CPS is characterized by the real-time progressions of physical quantities interleaved by the transition of discrete software and hardware states. HA are typically employed to model the behavior of CPS and to evaluate at design-time the correctness of the system, and its efficiency and robustness with respect to the desired safety requirements.

HA are called *safe* whenever given an initial set of states, the possible trajectories originated from these initial conditions are not able to reach a bad set of states. Proving a safety requirement requires indeed to solve a reachability analysis problem that is generally undecidable [27,188] for hybrid systems. However, this did not stop researchers to develop, in the last two decades, semi-decidable efficient reachability analysis techniques for particular classes of hybrid systems [14,29,101,118,119,163, 164,165,180,214].

Despite all this progress, the complexity to perform a precise reachability analysis of HS is still limited in practice to small problem instances (e.g., [26,27,28,188]). Furthermore, the models of the physical systems may be inaccurate or partially available. The same may happen when a CPS implementation employs third-party software components for which neither the source code or the model is available.

A more practical solution, close to testing, is to monitor and to predict CPS behaviors at simulation-time or at runtime [46]. The monitoring technology include the techniques to specify what we want to detect and to measure and how to instrument the system. Monitoring can be applied to:

- Real systems during their execution, where the behavioral observations are constructed from sensor readings.
- System models during their design, where the behaviors observed correspond to simulation traces.

In the following, we provide an overview of the main specification-based monitoring techniques available for CPS and HS. We also show the main applications of the monitoring techniques in system design and finally we discuss the main open challenges in this research field.

3.1 Context and Areas of Interest

To provide some context we first describe specification languages for hybrid systems, then discuss from specific issues of monitoring continuous and hybrid systems and then briefly present the state-of-the-art with respect to tools for monitoring these systems. Finally, we list applications of RV to hybrid systems.

3.1.1 Specification Languages

One of the main specification language that has been used in the research community for the formal specification of continuous and hybrid systems is *Signal Temporal Logic* (STL) [235,236]. STL extends *Metric Interval Temporal Logic* (MITL) [15], a dense-time specification formalism, with predicates over real-valued variables. This mild addition to MITL has an important consequence, despite its simplicity—the alphabet in the logic has an order and admits a natural notion of a distance metric. Given a numerical predicate over a real-valued variable and a variable valuation, we can henceforth answer the question on how far the valuation is from satisfying or violating the predicate. This rich feedback is in contrast to the classical yes/no answer that we typically get from reasoning about Boolean formulas. The quantitative property of numerical predicates can be extended to the temporal case, giving rise to the quantitative semantics for STL [143,133].

We can use with ease STL to specify real-time constraints and complex temporal relations between events occurring in continuous signals. These events can be trivial threshold crossings, but also more intricate patterns, identified by specific shapes and durations. We are typically struggling to provide elegant and precise description of such patterns in STL. We can also observe that these same patterns can be naturally specified with regular expressions, as time-constrained sequences (concatenations) of simple behavior descriptions.

Timed Regular Expressions (TRE) [25], a dense-time extension of regular expressions, seem to fit well our need of talking about continuous signal patterns. While admitting natural specification of patterns, regular expressions are terribly inadequate for specification of properties that need universal quantification over time. For instance, it is very difficult to express the classical requirement "every request is eventually followed by a grant" with conventional regular expressions (without negation and intersection operators). It follows that TRE complements STL, rather than replacing it.

CPS consist of software and physical components that are generally spatially distributed (e.g., smart grids, robotics teams) and networked at every scale. In such scenario, temporal logics may not be sufficient to capture not only time but also topological and spatial requirements. In the past five years, there has been a great effort to extend STL for expressing spatio-temporal requirements. Examples include *Spatial-Temporal Logic* (SpaTeL) [43,182], the *Signal Spatio-Temporal Logic* (SSTL) [252] and the *Spatio-Temporal Reach and Escape Logic* (STREL) [44].

3.1.2 Monitoring Continuous and Hybrid Systems

We first discuss some issues that are specific to the analysis of continuous and hybrid behaviors. We also provide an overview of different methods for monitoring STL with qualitative and quantitative semantics and matching TRE patterns.

Handling Numerical Predicates In order to implement monitoring and measuring procedures for STL and TRE, we need to address the problem of the computer representation of continuous and hybrid behaviors. Both STL and TRE have a dense-time interpretation of continuous behaviors which are assumed to be ideal mathematical objects. This is in contrast with the actual behaviors obtained from simulators or measurement devices and which are represented as a finite collection of value-timestamp pairs $(w(t), t)$, where $w(t)$ is the observed behavior. The values of w at two consecutive sample points t and t' do not precisely determine the values of w inside the interval (t, t'). To handle this issue pragmatically, interpolation can be used to "fill in" the missing values between consecutive samples. Some commonly used interpolations to interpreted sampled data are step and linear interpolation. Monitoring procedures are sensitive to the interpolation used.

Monitoring STL with Qualitative and Quantitative Semantics An offline monitoring procedure for STL properties with qualitative semantics is proposed in [236]. The procedure is recursive on the structure (parse-tree) of the formula, propagating the truth values upwards from input behaviors via super-formulas up to the main formula. In the same paper, the procedure is extended to an incremental version that computes the truth value of the sub-formulas along the observation of new sampling points.

There are several algorithms available in the literature for computing robustness degree of STL formulas [129,131,133,143,201,202,285]. The algorithm for computing the space robustness of a continuous behavior with respect to a STL specification was originally proposed in [143]. In [131], the authors develop a more efficient algorithm for measuring space robustness by using an optimal streaming algorithm to compute the min and the max of a numeric sequence over a sliding window and by rewriting the *timed until operator* as a conjunction of simpler *timed and untimed operators*. The procedure that combines monitoring of both space and time robustness is presented in [133].

Finally, the following two approaches have been proposed to monitor the space robustness of a signal with respect to an STL specification. The first approach proposed in [129] considers STL formulas with bounded future and unbounded past operators. The unbounded past operators are efficiently evaluated exploiting the fact that the unbounded history can be stored as a *summary* in a variable that is updated each time a new value of the signal becomes available. For the bounded future operators, the algorithm computes the number of look-ahead steps necessary to evaluate these operators and then uses a model to predict the future behavior of the system and to estimate its robustness. The second approach [125] computes instead an interval of robustness for STL formulas with bounded future operators.

Matching TRE Patterns An offline procedure for computing the set of all matches of a timed regular expression in a continuous or hybrid behavior was proposed in [308]. The procedure relies on the observation that any match set can always be represented as a finite union of two-dimensional zones, a special class of convex polytopes definable as the intersection of inequalities of the form $(x < a)$, $(x > a)$ and $(x - y < a)$. This algorithm has been recently extended to enable online matching of TRE patterns [309].

3.1.3 Tools

The following tools are publicly available and they support both the qualitative and the quantitative semantics for monitoring CPSs.

1. AMT 2.0 [255]: available at `http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/AMT/content.html`
2. Breach [130]: available at `https://github.com/decyphir/breach`
3. S-Taliro [17]: available at `https://sites.google.com/a/asu.edu/s-taliro/`
4. U-Check [81]: available at `https://github.com/dmilios/U-check`

The AMT 2.0 tool [255] provides a framework for the qualitative and quantitative analysis of xSTL, which is an extended Signal Temporal Logic that integrates TRE with STL requirements over analog system output signals. The software tool AMT is a standalone executable with a graphical interface enabling the user to specify xSTL properties, the signals and whether the analysis is going to be offline or incremental. The new version of the tool provides also the possibility to compute quantitative measurements over segments of the signals that match the properties specified using TRE [151]. AMT 2.0 offers also a *trace diagnostics* [150] mechanism that can be used to explain property violations.

Breach [130] and S-Taliro [17] are add-on Matlab toolboxes developed for black-box testing based verification [142] of Simulink/Stateflow models. These tools have also been used for other applications including parameter mining [332,328], falsification [2] to synthesis [278].

Finally, U-Check [81] is a stand-alone program written in Java, which deals with statistical model checking of STL formulas and parameter synthesis for stochastic models described as Continuous-Time Markov Chains.

3.1.4 Applications

Specification-based monitoring of cyber-physical systems (CPS) [254] has been a particularly fertile field for research on runtime verification leading to several theoretical and practical applications such as quantitative semantics, simulation-guided falsification, real-time online monitoring, system design and control. Here is an overview of the most relevant applications in the CPS scenario:

Real-time Monitoring of CPS. The complexity of the new generation of digital system-on-chip (SoC) and analog/mixed-signal systems (AMS) requires new efficient techniques to verify and to validate their behavior both at physical and software level. The simulation of such systems is now too time-consuming to be economically feasible. An alternative approach is to monitor the system under test (SUT) online by processing the signals and software traces that are observable after instrumentation [254]. This approach leverages the use of dedicated hardware accelerators such as *Field Programmable Gate Arrays* (FPGA) and of proper synthesis tools [200, 201, 297] that can translate temporal logic specifications into hardware monitors. This will be discussed in more detail in the next section dedicated to hardware supported runtime verification.

Falsification-based Testing. Specification-based monitoring is a very useful technique also at design-time. The engineers generally use MathWorks ™ Simulink[3] or OpenModelica[4] toolsets to model CPS functionalities. These models are complex hybrid systems that are very challenging to verify and test. *Falsification-based testing* [2, 3, 17, 18, 141, 253, 331] aims at automatically generating counter-examples that violate the desired requirements in a CPS model. This approach employs a formal specification language such as STL to specify the desired requirements, and a monitor (*the oracle*), that verifies each simulation trace for correctness against the requirement and it provides an indication as to how far the trace is from violation. For this reason, in the last decade there was a great effort to develop quantitative semantics for STL [259, 285, 284, 133, 11], where the binary satisfaction relation is replaced with a quantitative robustness degree function. The positive and negative sign of the robustness value indicates whether the formula is satisfied or violated, respectively. This quantitative interpretation can be exploited in combination with several heuristics (e.g., ant colony, gradient ascent, statistical emulation) to optimize the CPS design in order to satisfy or falsify a given formal requirement [2, 3, 17, 18, 45, 132, 141, 253, 331].

From Monitoring to Control Synthesis. The use of formal logic-based languages has also enabled control engineers to build tools that automatically synthesize controllers starting from a given specification [70]. Temporal logics such as Metric Temporal Logic (MTL) [215], and Signal Temporal Logic (STL) [235] have been employed to specify time-dependent tasks and constraints in many control system applications [17, 278, 321]. In the context of Model Predictive Control (MPC) [71, 213, 259, 277], the monitoring of temporal logics constraints over the simulated traces of a plant model can be used to find iteratively the input that will optimize the robustness for the specification over a finite-horizon.

3.2 Challenges

Although specification-based monitoring of CPS is a well-established research area, there are still many open challenges that need to be addressed. We now discuss some of the most important remaining challenges.

C 3.1. Autonomous CPS. The recent advances in machine learning (ML) has led to new fascinating artificial intelligence (AI) applications, such as autonomous CPS that can perceive, learn, decide and execute tasks independently, or with minimal human intervention in unpredictable environments. The lack of predictability, that results from using learning-enabled components, requires to think novel approaches for providing assurance. The main challenge is to develop new methods that go beyond the current state-of-the-art of RV technology to guarantee the trustworthiness of autonomous CPS by providing dynamic safety and security assurance mechanisms.

[3] `https://www.mathworks.com/products/simulink.html`

[4] `https://openmodelica.org/`

C 3.2. From design-time to runtime. Specification languages for CPS typically assume a perfect mathematical world in which time is continuous and the state variables are all observable with infinite precision. This level of abstraction is suitable to reason about CPS at the time of their design, where the system is modeled with differential equations and can be simulated with arbitrary precision and perfect observability. However, the passage from a CPS model to its implementation results in a number of effects that runtime monitors applied during the system operation need to take into account. For instance, CPS can be only observed at sampled points in time, some state variables may not be observable and the sensors may introduce noise and inaccuracies into measurements, including sampling noise. As a consequence, there is an urgent need to address these questions in the context of runtime verification of CPS.

C 3.3. Limited resources. CPS introduce some specific constraints on available resources that need to be taken into account by runtime verification solutions. CPS are reactive systems operating at a certain frequency, hence the monitor needs to operate at least at the same speed as the system. In contrast to classical software, instrumentation of some components in the CPS can be hard or impossible. It follows that runtime monitors may need to rely on partially observable streams of data. CPS are often safety-critical and have hard timing constraints. As a consequence, runtime monitors must not alter the timing-related behavior of the observed system. Developing monitoring solutions that take into consideration specific limitations of CPS remains an important challenge that needs still to be properly addressed.

C 3.4. From real-time to spatial and spectral specifications. Most of the existing work on runtime monitoring of CPS is focused on real-time temporal properties. However, CPS often consist of networked spatially distributed entities where timing constraints are combined with spatial relations between the components. In addition, many basic properties of continuous CPS entities are naturally definable in spectral (for instance frequency) domain [97,134]. There is a necessity to study specification formalisms that gracefully integrate these important CPS aspects.

C 3.5. Fault-localisation and explanation. Detecting a fault while monitoring a CPS during its design or deployment time involves understanding and correcting the error. Complementing runtime verification methods with (semi) automated fault localisation [48] and explanation could significantly reduce the debugging efforts and help the engineer in building a safe and secure system.

4 Hardware

Hardware supported runtime verification (HRV) studies how to use hardware to build dynamic solutions for reliabilty assesment. The goal is to alleviate the extensive analysis required for complex designs, by shifting from offline and limited data sets to online simultaneous non-intrusive analysis. The use of hardware brings an immense potential for runtime observation and can even allow the continuous assessment of the behavior exhibited by the system. Observation and simultaneous correctness checking of system internals can reach a level of detail that is orders of magnitude better than today's tools and systems provide. Note that the use of "hardware" in HRV refers to the use of hardware as an element of the RV solution, even though the system under study can also be analyzed at a low-level that includes hardware characteristics.

Online runtime verification hardware-based approaches may take advantage of multiple technologies, for example, hardware description languages and reconfigurable hardware. The combination of these technologies provides the means for observability, non-intrusiveness, feasibility, expressiveness, flexibility, adaptability and responsiveness of hardware-based monitors that observe and monitor a target system and allow to react to erroneous behavior. In addition, HRV can be used for other analysis, such as performance monitoring.

Several solutions have been proposed that approach runtime verification (RV) differently, diverging on the methodologies used, goals and target system-induced limitations. Whether the monitor executes on external hardware or on-system, what the monitor watches (that is, the meaningful events it cares about: the events of interest), how it is connected to the system and what is instrumented or not, are dependent on both the characteristics of the system being monitored and the goals of the monitoring process.

4.1 Context and Areas of Interest

To present the context of HRV in order to later describe the challenges, we describe the following aspects separately: the pursue of non-intrusiveness monitoring, the study of the feasibility and limitations of hardware-based monitoring, the landscape of design approaches and the flexibility. We finally list some existing use cases.

4.1.1 Non-intrusiveness

Ideally, observing and monitoring components should not interfere with the normal behavior of the system being observed, thus negating what is called "the observer effect" or "the probe effect" [167], in which the observing methodology hinders the system behavior by affecting some of its functional or non-functional (e.g., timeliness) properties. Hardware-based approaches are inherently non-intrusive, while software-based solutions normally exhibit some degree of intrusiveness, even if minimal. Therefore, it is widely acknowledged that these approaches must be used with care.

For example, the delays implicitly associated with the insertion of software-based probes may ill affect the timing and synchronisation characteristics of concurrent programs. Moreover, and perhaps less intuitively, the removal of such probes from real-time embedded software which, in principle, leads to shorter program/task execution times and may render a given task set unschedulable due to changes in the corresponding cache-miss profile [233, 249, 320]. Non-intrusiveness, i.e. the absence of interference may then be referred to as a RV constraint. RV constraints are not only relevant, but in fact fundamental, for highly critical systems [269].

A comprehensive overview of various hardware (including on-chip), software and hybrid (i.e., a combination of hardware and software) methodologies for system observation, monitoring and verification of software execution in runtime is provided in [316].

System observing solutions can be designed to be directly connected to some form of system bus, enabling information gathering regarding events of interest, such as data transfers and signalling taking place inside the computing platform, namely instruction fetch, memory read/write cycles and interrupt requests, with no required changes on the target system's architecture. Examples of such kind of hardware-based observation approaches are proposed in [208,266,271,281].

As emphasized in [316] observing mechanisms should: (1) be minimally intrusive, or preferably completely non-intrusive, so as to respect the RV constraint; (2) provide enough information about the target system so that the objectives of runtime verification can be met.

4.1.2 Observability

Another important aspect raised in [316] is the occasional limited observability of program execution with respect to its internal state and data information. In general, software-based monitoring may have access to extensive information about the operation of a complex system, in contrast to the limited information available to hardware probes [316].

Thus, one first challenge is that hardware-based probes must be capable of observing enough information about the internal operation of the system to fulfil the purpose of the monitoring [316]. Gaining access to certain states or information is often problematic, since most systems do not provide access to system operation and software execution details. So, observability is sometimes limited to the data made available or accessible to observing components. Low observability of target system operation affects not only traditional hardware monitors, but also may jeopardize hybrid monitoring and may deem these observing and monitoring techniques ineffective.

4.1.3 Feasibility

General purpose *Commercial Off-The-Shelf* (COTS) platforms offer limited observing and monitoring capabilities. For example, in those platforms based on Intel x86 architectures observability is restricted to the Intel Control Flow Integrity [195] and to the Intel Processor Trace [283] facilities. Trying to enhance system observability through physical probing implies either a considerable engineering effort [210] or is restricted to specific behaviors, such as input/output operations [266].

The trend to integrate the processing entities together with other functional modules of a computing platform in an *Application Specific Integrated Circuit* (ASIC), often known as *System on a Chip* (SoC), can dramatically affect the overall system observability, depending on whether or not special-purpose observers are also integrated.

The shortcomings and limitations of debug and trace resources regarding runtime system observation is analyzed in [223], concluding that the deep integration of software and hardware components within SoC-based devices hinders the use of conventional analysis methods to observe and monitor the internal state of those components. The situation is further exacerbated whenever physical access to the trace interfaces is unavailable, infeasible or cost prohibitive.

With the increased popularity of SoC-based platforms, one of the first on-chip approaches to SoC observability was introduced in [301], where the authors presented MAMon, a hardware-based probe-unit integrated within the SoC and connected via a parallel-port link to a host-based monitoring tool environment that performs both logic-level (e.g., interrupt request assertion detection) and system-level (e.g., system call invocation) monitoring. This approach can either be passive (by listening to logic- or system-level events) or activated by (minimally intrusive) code instrumentation.

Many SoC designs integrate modules made from Intellectual Property (IP) cores. An IP core design is pre-verified against its functional specification, for example through assertion-based verification methods. In hardware-based designs, assertions are typically written in verification languages such as the *Property Specification Language* (PSL) [194] and *System Verilog Assertions* (SVA) [193]. The pre-verification of IP core designs contributes to reduce the effort placed in the debug and test of the system integration cycle.

The work [306] presents an in-circuit RV solution that targets the monitoring of the hardware itself rather than software. Runtime verification is done by means of in-circuit temporal logic-based monitors. Design specifications are separated into compile-time and runtime properties, where runtime properties cannot be verified at compile-time, since they depend on runtime data. Compile-time properties are checked by symbolic simulation. Runtime properties are verified by hardware monitors being able to run at the same speed as the circuits they monitor.

System-wide observation of IP core functionality requires the specification of a set of events to be observed and a set of observation probes. The IP core designer will be the best source of knowledge for determining which event probes can provide the highest level of observability for each core. Such kind of approach is followed in [222], for the specification of a low-level hardware observability interface: a separate dedicated hardware observability bus is used for accessing the hardware observation interface.

The approach described in [222] was further extended in [223] to include system level observations, achieved through the use of processor trace interfaces. The solution discussed in [223] introduces a System-level Observation Framework (SOF) that monitors hardware and software events by inserting additional logic within hardware cores and by listening to processor trace ports. The proposed SOF provides visibility for monitoring complex execution behavior of software applications without affecting the system execution. Engineering and evaluation of such approaches has resorted to FPGA-based prototyping [222,223].

Support for such kind of observation can be found also in modern processor architectures with multiple cores, implemented as single chip solutions and natively integrating embedded on-chip special-purpose observation resources, such as the ARM CoreSight [22,256].

4.1.4 Design approaches

Nowadays there are two approaches for embedded multicore processor observation. Software instrumentation is easy to use, but very limited for debugging and testing (especially for integration tests and higher levels). A more sophisticated approach and key element in multicore observation are embedded trace based emulators. A special hardware unit observes the processor's internal states, compresses and outputs this information via a dedicated trace port. An external trace device records the trace data stream and forwards the data after the observation period to, e.g. a personal computer for offline decompression and processing. Unfortunately, this approach still suffers from serious limitations in trace data recording and offline processing:

– Trace trigger conditions are limited and fixed to the sparse functionality implemented in the "embedded trace" unit.
– Because of the high trace data bandwidth it is impracticable on today's storage systems to save all the data obtained during an arbitrary long observation.

– There is a discrepancy between trace data output bandwidth and trace data processing bandwidth, which is usually several orders of magnitude slower. This results in a very short observation period and a long trace data processing time, which renders the debugging process inefficient.

Hardware supporting online runtime verification could overcome these limitations. Trace data is not stored before being pre-processed and verified, because both are done online. Debugging and runtime verification are accomplished without any noticeable interference with the original system execution. Verification is based on a given specification of the system's correct behavior. In case a misbehavior is detected, further complex processing steps are triggered. This challenging solution enables an autonomous, arbitrary enduring observation and brings out the highest possible observability from "embedded trace" implementations.

Other solutions place the observation hardware inside the processing units, which may, in some situations, require their modification. Some simple modifications may enable lower-level and finer-grained monitoring, for example by allowing the precise instant of an instruction execution to be observed. The choice of where to connect a runtime verification hardware depends on the sort of verification one aims to perform and at which cost, being a design challenge.

A *Non-Intrusive Runtime Verification* (NIRV) observer architecture for real-time SoC-based embedded systems is presented in [271]. The observer (also called *Observer Entity*, OE) synchronously monitors the SoC bus, comparing the values being exchanged in the bus with a set of configured observation points, the events of interest. Upon detection of an event of interest, the OE time-stamps the event and sends it an external monitor. This approach is extended in [177] to enforce system safety and security using a more precise observation of programs execution, which are secured through the (non-intrusive) observation of the buses between the processor and the L1 cache sub-system.

A wide spectrum of both functional and non-functional properties can be targeted by these RV approaches, from timeliness to safety and security, preventing misbehavior overall. The effectiveness of system observability is crucial for securing the overall system monitoring. Hardware-based observation is advantageous given its non-intrusiveness, but software-based observation is more flexible, namely with respect to capturing of context-related data.

4.1.5 Flexibility: (self-)adaptability and reconfiguration

Requirements for (self-)adaptability to different operational conditions call for observers (and monitors) flexibility, which may be characterized by a ready capability to adapt to new, different, or changing needs. Flexibility implies that observing resources should be re-configurable in terms of the types and nature of event triggers. This configurability may be defined via configuration files, supported online by self-learning modules, or a combination of both. Reconfigurable hardware implementations usually provide sufficient flexibility to allow for changes of the monitored specification without re-synthesising the hardware infrastructure. This is a fundamental characteristic since logic synthesis is a very time-consuming task and therefore unfit to be performed online. Observer and monitor reconfigurability can be obtained in the following ways:

– Using reconfiguration registers that can be changed online [271], a flexible characteristic that supports simple to moderate adaptability capabilities. Examples include to redefine the address scope for a function stack frame, upon its call, or to define function's calling addresses upon dynamic linking with shared object libraries.

- Selecting an active monitor or a monitor specification from a predefined set of mutually exclusive monitors [286]. This corresponds to a mode change in the operation of the system. Mode changes needs to secure overall system stable operations [267].
- Using a reconfigurable single monitor [276], which allows to update the monitor through the partial reconfiguration capabilities enabled by modern FPGAs.

The approach in [276] implements intrusion detection in embedded systems by detecting behavioral differences between the correct system and the malware. The system is implemented using FPGA logic to enable the detection process to be regularly updated and adapt to new malware and changing system behavior. The idea is to protect against the execution of code that is different from the correct code the system designer intends to execute. The technique uses hardware support to enable attack detection in real time, using finite state machines.

System adaptation triggered by non-intrusive RV techniques is approached in [286] for complex systems, such as *Time- and Space-Partitioned* (TSP) systems, where each partition hosts a (real-time) operating system and the corresponding applications. Special-purpose hardware resources provide support for: partition scheduling, which are verified in runtime through (minimally intrusive) RV software; process deadline violation monitoring, which is fully non-intrusive while deadlines are fulfilled. Process level exception handlers, defined the application programmer, establish the actions to be executed by software components when a process deadline violation is detected. The monitoring component which analyzes the observed events (the trace data) may be a component belonging to RV hardware itself, checking the system behavior as it observes.

4.1.6 Use case examples

Given the numerous possibilities for implementing RV in hardware, multiple contributions have been made that tackle the ongoing search for improvement of hardware-based RV monitors. Some solutions address monitoring and verification in a single instance [281]. Here, the verification procedure is mapped into soft-microcontroller units, embedded within the design, and use formal languages such as past-time Linear Temporal Logic (ptLTL). An embedded CPU is responsible for checking ptLTL clauses in a software-oriented fashion.

A System Health Management technique was introduced in [282] which empowers real-time assessment of the system status with respect to temporal-logic-based specifications and also supports statistical reasoning to estimate its health at runtime. By seamlessly intercepting sensor values through read-only observations of the system bus and by on-boarding their platform (rt-R2U2) aboard an existing FPGA already built into the standard UAS (Unmanned Aerial Systems) design, system integration problems of software instrumentation or added hardware were avoided, as well as intrusiveness.

A runtime verification architecture for monitoring safety critical embedded systems which uses an external bus monitor connected to the target system, is presented in [207]. This architecture was designed for distributed systems with broadcast buses and black-box components, a common architecture in modern ground vehicles. This approach uses a passive external monitor which lines up well against the constraints imposed by safety-critical embedded systems. Isolating the monitor from the target system helps ensure that system functionality and performance is not compromised by the inclusion of the monitor.

The use of a hardware-based NIRV approach for mission-level adaptation in unmanned space and aerial vehicles is addressed in [287] with the goal to contribute to mission/vehicle survivability. For each phase of a flight, different schedules are defined to three modes: normal, survival, recovery. The available processor time is allocated to the different vehicle functions accordingly with its relevance within each mode: normal implies the execution of the activities defined for the mission; survival means the processor time is mostly assigned to fundamental avionic functions; recovery foresees also the execution of fault detection, isolation and recovery functions.

Gouveia and Rufino [177] attack the problem of fine-grained memory protection in cyber-physical systems using a hardware-based observation and monitoring entity are presented. To ensure the security of the observer itself, the monitor is designed as a black box, allowing it to be viewed in terms of its input and output but not its internal functioning and thus preventing malicious entities from hijacking its behavior.

No previous study concerning hardware-based observability has tackled the problem of applying the concepts and techniques to the non-intrusive observation and monitoring of programs in interpreted languages, such as Python and Java bytecode, running on the corresponding virtual machines.

4.2 Challenges

C 4.1. Observability. There is no general results on defining which hardware entities (system bus, processor internal buses, IP core internals) of a system should be instrumented to guarantee the required observability and how to probe such entities. In general, observation at different levels of abstraction should be supported, from logic-level events (e.g., interrupt, request, assertion) up to system (e.g., system call invocation) and application levels (e.g., value assigned to a given variable).

C 4.2. Effectiveness. To ensure that hardware-based probing is able to provide effective system observability, meaning all the events of interest should be captured, while maintaining the complexity of hardware instrumentation in conformity with SWaP (Size, Weight and Power) constraints. This is especially important for observation and monitoring of hardware components, where the RV resources should have a much lower complexity than the observed infrastructure, but this results could also be applicable to the monitoring of software components.

C 4.3. Feasibility and flexibility. To handle the potentially high volumes of trace data produced by extensive system observation is challenge. It includes confining the observed events of interest, and the use of advanced compression, pre-processing and runtime verification techniques to reduce the gap between trace data output and trace data processing capabilities. Also, mapping of formal specification of system properties into actual observing and monitoring actions, making use of a minimal set of highly effective hardware/software probing components and monitors. If applicable, provide support for flexible observation and monitoring, thus opening room for the integration of RV techniques in (self-)adaptable and reconfigurable systems.

C 4.4. Hybrid approaches for observability. Combining software-based instrumentation with hardware-based observability in a highly effective hybrid approach, to:

(1) Capture program execution flows and timing, without the need for special-purpose software hooks; (2) Observe fine-grained data, such as read/write accesses to global and local variables; (3) Monitor bulk data (e.g. arrays) through the observation of read/write accesses to individual members.

C 4.5. Advanced system architectures. Extending hardware-based observability to advanced system architectures, such as processor and memory virtualisation, including time- and space-partitioning, and also to the execution of interpreted languages including bytecode that runs on virtual machines, like JVM.

5 Security and Privacy

In the last years there has been a huge explosion in the availability of large volumes of data. Large integrated datasets can potentially provide a much deeper understanding of both nature and society and open up many new avenues of research. These datasets are critical for addressing key societal problems—from offering personalized services, improving public health and managing natural resources intelligently to designing better cities and coping with climate change. More and more applications are deployed in our smart devices and used by our browsers in order to offer better services. However, this comes at a price: on one side most services are offered in exchange of personal data, but on the other side the complexity of the interactions of such applications and services makes it difficult to understand and track what these applications have access to, and what they do with the users' data. Privacy and security are thus at stake.

Cybersecurity is not just a buzzword, as stated in the recent article "All IT Jobs Are Cybersecurity Jobs Now" [240] where it is said that "The rise of cyberthreats means that the people once assigned to setting up computers and email servers must now treat security as top priority". Also, "The largest ransom-ware infection in history" [304]. Referring to the event above, the Europol chief stated in a recent BBC interview that "Cybersecurity should be a top line executive priority and you need to do something to protect yourself" [69].

Besides the above examples, which are well-known given their massive impact in the media and society, we know that security and privacy issues are present in our daily lives in different forms, including botnets, distributed denial-of-service attacks (DDoS), hacking, malware, pharming, phishing, ransomware, spam, and numerous attacks leaking private information [299]. The (global) protection starts with the protection of each single computer or device connected to the Internet. However, nowadays only partial solutions can be done statically. Runtime monitoring, verification and enforcement are thus crucial to help in the fight against security and privacy threats.

Remark. Given the breadth of the Security & Privacy domain, we do not present an exhaustive analysis of the different application areas. We deliberately focus our attention on a small subset of the whole research area, mainly privacy concerns from the EU General Data Protection Regulation (GDPR), information flow, malware detection, browser extensions, and privacy and security policies. Even within those specific areas, we present a subset of challenges emerging from this areas.

5.1 Context and Areas of Interest

We present now the context and state-of-the-art of monitoring in the following security relate sub-areas: GDPR, information flow, malware detection, browser extensions and privacy and security policies.

5.1.1 GDPR (General Data Protection Regulation)

The European *General Data Protection Regulation* [117] (GDPR)—which as adopted on 27 April 2016 and entered into application on 25 May 2018—subjects companies, governmental organizations and any other data collector to stringent obligations when it comes to user privacy in their digital products and services. Consequently, new systems need to be designed with privacy in mind (*privacy-by-design* [96]) and existing systems have to provide evidence about their compliance with the new GDPR rules. This is mandatory, and sanctions for data breaches are tough and costly.

As an example, Article 5 of GDPR, related to the so-called *data minimization principle*, states: "Personal data must be adequate, relevant, and limited to the minimum necessary in relation to the purposes for which they are processed". While determining what is "adequate" and "relevant" might seem difficult given the inherent imprecision of the terms, identifying what is "minimum necessary in relation to the purpose" is easier to define and reason about formally.

Independently on whether we are considering privacy by design or giving evidence about privacy compliance for already deployed systems, there are some issues to be considered. Not all the obligations stated in the regulations can be easily translated into technical solutions, so there is a need to identify which regulations are enforceable by technical means. For those rules or principles identified as being enforceable by software, it is hard for engineers to assess and provide evidence of whether a technical design is compliant with the law due to the gap existing between a legal document written in natural language and a technical solution in the form of a software system.

Consider again the data minimization principle. One way to understand minimization is on how the data is *used*, that is we could consider ways to identify the *purpose* for which the input data collected is used in the program. In this case we would need to look inside the program and track the usage of the data by performing static analysis techniques like tainting, def-use, information flow, etc. This, in turn, requires a precise definition of what "purpose" means and a way to check that the intended purpose matches the real actions that the program take to process the data at runtime. Another aspect of minimization is related to when and how the data is *collected* in order to limit the collection of data to what is actually needed to perform the purpose of the program. In this case we could consider that the purpose is given by the specification of the program, which is the approach followed by Antignac et al. [19]. This results indicate that it may be possible to enforce data minimization at runtime, at least in what concerns some of its aspects. But other privacy principles are more difficult to tackle.

5.1.2 Information Flow

In computer systems, it is often necessary to prevent some objects to access specific data. These permissions are usually defined through security policies, and enforced using access control mechanisms. However, such mechanisms are typically insufficient in practice. For instance, an application could require to access both private data—such as the user contact list—and to connect to Internet but, once the application is granted by the operating system's access control policy, one would like to ensure that no data from the contact list (assumed to be confidential) leaks to the Internet (a public channel). Enforcing such fine-grained security policies require information flow control mechanisms. These mechanisms allow untrusted applications to access confidential data as soon as they do not leak these data to public channels. Denning's seminal work [122, 123] in that field proposed static verification techniques to ensure that a program does not leak any confidential data. This property is usually called *non-interference*, first formalized by Goguen and Meseguer [175]. More generally, non-interference states that no private data leaks to a public channel, either directly or indirectly. An indirect non-secure flow may appear for instance when two different values of some public data may be emitted on a public channel depending on some private conditions. In this case, an observer can infer part of the private information just by observing public data. From the eighties to the early 2000's, many efforts have been put in verifying non-interference properties statically [290,315].

In 2004 Vachharajani et al. [310] abandoned static approaches and proposed Rifle, a runtime information flow security system. After that, dynamic information flow approaches have been proposed for different settings (*e.g.* JavaScript [34], or applied to databases [333]). The main advantage of dynamic information flow is its ability to deal with dynamic languages and dynamic security policies. It is also usually more permissive than static approaches with respect to non-interference: dynamic approaches may accept secure flows that would be rejected statically. However, pure dynamic approaches have a major drawback: they cannot take into account the branches uncovered by the examined executions and so they may miss (indirect) insecure flows. In particular, Russo and Sabelfeld [289] demonstrated that pure dynamic approaches cannot be sound with respect to flow-sensitive non-interference, in the form of Hunt and Sands [192]. However flow-sensitivity is a very useful feature in practice, since it is more permissive than flow-insensitivity by accepting that memory locations store values of different security level.

In 2006 Le Guernic et al. [221] proposed a hybrid approach that combines soundness of a static approach and permissiveness of a dynamic approach. In recent years, hybrid information flow has received a lot of attention, for instance for languages such as C [41], Haskell [83], and JavaScript [294,186]. To deal with the unsoundness of dynamic approaches, it is also possible to consider multiple executions [126] or multiple facets [35], the latter consisting in mapping a variable to several values (or facets), each of them corresponding to a particular security level.

Different variants of non-interference and ways of verifying them are described by Hedin and Sabelfeld's [187] and by Bielova and Rezk [78].

5.1.3 Malware Detection and Analysis

Malware refers to a malicious software specifically designed to disrupt, damage, or gain unauthorized access to a computer system. Malware usually exploits specific system vulnerabilities, such as a programming bug in software (e.g., a browser application plugin) or a bug in the underlying platform or OS. Malware infiltration effects range from simple disruption of the proper behavior of the system to destruction or theft of private and sensitive data. The huge number of devices interconnected through the

Internet has turned the infection of malware a very serious threat, even more with the current trend of digitizing almost all human activities, notably economical transactions.

Malware *detection* is concerned with identifying software that is potentially malicious, ideally before the malware acts destructively. Malware *analysis* is about identifying the true intent and capabilities of malware by looking at some aspects of the code (statically) or by running it (dynamically).

Static analysis examines malware with or without viewing the actual code. The technical indicators gathered with basic static analysis can include file name, hashes, file type, file size and recognition by using tools like antivirus. When it is possible to inspect the source code, static malware analyzers try to detect whether the code has been intentionally obfuscated or try to identify concrete well-known malicious lines of code. Dynamic analysis, on the other hand, runs the malware in a controlled environment to observe its behavior, in order to understand its functionality and identify indicators of potential danger. These indicators include domain names, IP addresses, file path locations, and whether there are additional files located on the system. See [198,292, 299,334] for surveys on malware detection techniques.

5.1.4 Browser Extensions

Browser extensions are small applications executed in a browser context in order to provide additional capabilities and enrich the user experience while surfing the web. The acceptance of extensions in current browsers is unquestionable. For instance, as of 2018, Chrome's official extension repository has more than 140,000 applications, with some of these extensions having more than 10 million users. When an extension is installed, the browser often pops up a message showing the permissions that this new extension requests and, upon user approval, the extension is then installed and integrated within the browser. Extensions run through the JavaScript event listener system. An extension can subscribe to a set of events associated with the browser (e.g., when a new tab is opened or a new bookmark is added) or the content (e.g., when a user clicks on an HTML element or when the page is loaded). When a JavaScript event is triggered, the event is captured by the browser engine and all extensions subscribed to this event are executed.

Research on the understanding of browser extensions, detecting possible privacy and security threats, and mitigating them is on its infancy. The potential danger of extensions has been highlighted in [191] where extensions were identified to be "the most dangerous code to user privacy" in today's browsers. Some recent works have focused on tracking the provenance of web content at the level of DOM (Document Object Model) elements [24].

Another relevant issue is the order in which extensions are executed. When installed, extensions are pushed to an internal stack within the browser, which implies that the last installed extension is the last one that will be executed.

Recent works [268] demonstrates empirically that this order could be exploited by an unprivileged malicious extension (i.e., one with no more permissions than those already assigned when accessing web content) to get access to any private information that other extensions have previously introduced. To the best of our knowledge, there still is no solution to this problem.

Finally, there is the problem of collusion attacks, which occurs when two or more extensions collaborate to extract more information from the user based on the individual permissions of each extension. Even tough in isolation they cannot do any harm, they can exercise an additional power by collaborating and combining their privileges. With few exceptions [291], this is an unexplored area.

Given that extensions may subscribe to events after they have been installed (i.e., at runtime), there is no way to statically detect potential attacks.[5] One of the few works providing a runtime solution to information flow in browsers (Chromium in particular) is [68].

Overall, there still are concerns regarding the effect of browser extensions on security and privacy. Giving the limitations on what can be obtained by static analysis, solutions to mitigate these issues must be accomplished by means of runtime monitoring techniques.

5.1.5 Privacy and Security Policies

One way to mitigate security and privacy threats is to have suitable and powerful policies which are enforced statically or at runtime. This, however, is not easy for different reasons. First, defining precisely a policy language requires to introduce its syntax (what the policies can talk about), characterize its scope (what are the limitations, i.e., what cannot be expressed/captured by the language), and define an enforcement mechanism (how to implement the mechanism that ensures the policies are to be respected). Getting a sound and complete result is too restrictive in general. Second, static policies may be enforced only in very specific cases and have to be done by designers and programmers at a very early stage of the software development process. In some cases, this may be done at runtime when the code is downloaded, but it requires to isolate the code to perform the analysis, which is not always possible. Last, security and privacy policies could be enforced at runtime: by mitigating the attack right after it is detected. This is not possible in general as we cannot foresee all possible future threats and sometimes when an attack is detected, it is usually too late.

5.2 Challenges

C 5.1. Monitoring GDPR. One of the main challenges is to identify which privacy principles might be verified or enforced by using monitors. As the regulation is quite extensive, we advocate to start with the principle of *data minimization* as an example of the kind of challenges the community might face.

C 5.2. Monitoring Data Minimization. When considering how the data is *used*, a challenge is that we will not be able to do runtime verification in a black box manner. Getting access to proprietary code can be an issue. Concerning when and how the data is *collected*, we could do runtime verification in a black box manner, but data minimization is not monitorable in general [270]. For the more general notion of distributed data minimization, the property is not monitorable, therefore new techniques using *grey box* runtime verification might be needed [80].

C 5.3. Hybrid Information Flow. As mentioned earlier, it is not possible to have a sound yet permissive dynamic information flow analysis [289]. Therefore, an important challenge for information flow monitoring is the design of a hybrid (static/dynamic)

[5] Extensions may statically declare to which events they want to subscribe, but there is nothing forbidding them to subscribe to new events later at runtime.

mechanism that is efficient yet permissive, and that can deal with real programs and security policy.

C 5.4. Monitoring Declassification and Quantitative Information Flow. Non-interference is often too strong a property. For instance, a password checker usually leaks one bit of information: whether the password is correct. Declassification and quantitative information flow aim to solve this issue, but verifying these properties is very hard. In spite of some initial work on hybrid approaches [74], monitoring these properties remains an unresolved challenge.

C 5.5. Generic Language for Information Flow. There are many variants and flavors of important properties like non-interference, but there is currently no main-stream accepted language that encompasses all these security policies, which are now recognized to be hyper-properties [102]. The challenge is the design and adoption of a formalism for the hyperproperties of interest in information flow security and the thorough study of its monitoring algorithms and limitations.

C 5.6. Browser extensions. One challenge on the enforcement side is how to ensure that malicious extensions do not expose private information from a user's homepage. This private leakage might be done by an external entity or by another extension which may aggregate this information with the information the extension has already collected, eventually performing a collusion attack. A related issue has to do with implementation: a robust runtime enforcement mechanism might need to modify the core of the browser (e.g., Chromium), which is quite invasive and requires a high level of expertise.

C 5.7. Privacy and security policies. One challenge is how to define security and privacy policy languages to write policies about concrete known threats. Also, this challenge involves the use of runtime monitoring techniques in order to detect potential and real threats, log that information and give this to an offline analyzer to identify patterns in order to generalize existing policies, or create new ones. A related challenge is how to learn the policies at runtime. This could be done by learning them from the attacker models (e.g., as in [1]), and improve the precision taking feedback from the runtime monitors.

6 Reliable Transactional Systems

The human society is increasingly dependent on computing systems, even in areas like entertainment (e.g., Netflix), social (e.g., Facebook) and economic interactions (e.g., Amazon). The ubiquity of computer systems, and the large scale at which they operate, make hardware and software failures both common and inevitable. At first glance it might seem that the majority of systems should not experience failures as frequently because they do not serve a world-scale user base. But with the advent of Infrastructure as a Service (IaaS) products (e.g., Amazon EC2) small and medium-sized companies are deploying their systems over IaaS offerings [23], which are supported by fault prone large-scale clusters [174]. This setting exploits modern hardware systems features to provide fault tolerance while keeping the software systems running efficiently, correctly, and with ease to develop and use, hence building computer systems with improved reliability and resilience and lower energy consumption.

Database systems have successfully exploited parallelism for decades, both at software and hardware levels. Databases can improve their performance by issuing many queries simultaneously and by running those queries on multiple computing systems in parallel, while preserving the same programming model as if the queries were executed one at a time in a single computing system. Transactions are at the core of most database systems. A transaction is an abstraction that specifies a program semantics where computations behave as if they are executing one at a time with exclusive access to the database. Transactional systems implement a *serializable* model. This means that even if the system allows multiple transactions to execute concurrently, the final result of their execution must be indistinguishable from executing one after the other (in some total order). Consequently, a transaction is a sequence of actions that appear to execute instantaneously as a single, indivisible, operation. The transactional system manages concurrency between transactions automatically, and is free to execute transactions concurrently as long as the result is equivalent to some serial execution of the transactions.

State machine replication (SMR) [219, 295] is the standard way to build such fault-tolerant systems. An SMR system maintains multiple replicas that keep a copy of the system's data, and coordinates the execution of operations on each of those data replicas. Since replicas also execute every operation submitted to the system, the system can continue operating as long as a majority of correct replicas execute the operations. When requests to execute operations arrive, an "agree-execute" protocol keeps replicas synchronized: they first agree on an order to execute the incoming operations, and then execute the operations sequentially in the agreed order, driving all replicas to the same final state. However, to take advantage of contemporary hardware systems, one should use all the available processor cores to execute multiple operations at the same time. That said, this concurrent execution of operations is at odds with the "agree-execute" protocol because concurrent execution is inherently non-deterministic so replicas may arrive at different final states and the system could become inconsistent.

Improving SMR's efficiency and performance can be achieved by exploiting multi-core processors, while still preserving determinism and correctness. This, however, requires to have operations that can be expressed as serializable transactions, and that the concurrency control protocol ensures that the concurrent execution of transactions respects the order replicas have agreed upon.

In a typical SMR setting, a set of clients concurrently submit requests to the system. The system, made of replicas, runs an agreement protocol, e.g., Paxos [220], that totally orders the incoming requests. Each replica executes the requests sequentially in the agreed order, driving all the (correct) replicas to the same final state. Essentially, we can divide state machine replication in two phases. First, the *agreement phase*, where replicas agree on an order for all requests. This is then followed by the *execution phase*, where replicas execute the requested operations in the agreed order. When using SMR there is a clear tension between the fact that the replicas have multi-core processors and the requirement that replicas execute the operations in a specific order.

Recovery and reparations in transactional systems [107] are multi-layered: when recovering within a transaction which may still succeed, reparations may be expressed in a *try-catch* fashion. However, if the action is considered to have failed, then any previously completed parts of the transaction need to be rolled back. This is done to preserve the atomicity of the transaction, i.e., either the transaction entirely succeeds or entirely fails. The problem arises when it is not possible to isolate a transaction with the result that its actions affect other parts of the system before the transaction is committed. This usually happens due to the long-life nature of the transaction — making it infeasible to lock the relevant resources for a long duration.

6.1 Context and Areas of Interest

Transactional systems cover a broad area. To later present challenges to RV, we describe here some of the important aspects, in particular dependable storage, coordination services, network services and memory contention management.

6.1.1 Dependable Storage Systems

Main database vendors, such as IBM and Oracle, have business solutions for high-performant dependable storage systems. Innovative approaches to such dependable storage systems are based on state machine replication, either in KV-stores [73,197, 300], filesystems [95,227], or transactional storages [139,169]. These systems are frequently used to build business-critical (and sometimes even life-critical) systems and must be constantly monitored to assess the correct behavior of the storage system. Monitoring these systems, specially those involving SMR, is challenging, as it allies the challenges of monitoring distributed systems with the challenges of monitoring transactional systems, both in terms of the architecture of monitoring system itself and of the information to be collected to reason upon [203,75].

6.1.2 Coordination services

Concurrent operations on distributed applications frequently need to be coordinated to ensure system correctness, otherwise the operations may be executed out-of-order or, in the case of SMR, the nodes may diverge and render the system inconsistent. These services are often provided by a small database, which stores configuration data to implement resource locking, leader election, message ordering, etc. Such coordination systems have been recently used in more complex solutions, for example in: i) Google's Chubby distributed lock service [86], which is used by Bigtable (now in production in Google Analytics and other products); ii) the Ceph storage system [318], where the coordination system is part of the monitor processes to agree which OSDs are up and in the cluster; iii) the Clustrix distributed SQL database, which leverages on a coordination system for distributed transaction resolution. A monitor for such systems must incorporate the complexities of the coordination/decision rules and of the control system itself.

6.1.3 Network Services

Software-Defined Networks (SDNs) are a step towards the separation of the network control and data planes, aiming at improving the manageability, programmability and extensibility of computer networks. In these SDNs, the controller should neither be a bottleneck nor a single point of failure. State machine replication is a natural answer to such fault-tolerance requirements. For example, the Ananta distributed load balancer [265] uses Paxos for maintaining high-availability in its manager component and serves thousands of data flows per day in the Windows Azure cloud. Such network services are transparently used by applications running in the cloud, and are yet another example of a SMR system, with the same monitoring requirements.

6.1.4 Main Memory Contention Management

The transactional model as used by database systems can be of use to manage the contention to shared data residing in main memory. This was first observed by Lomet in 1977 [228], and proposed as a hardware solution by Herlihy and Moss in 1993 [190], and by Herlihy et al. in 2003 [189] as the first practical software only solution. Some programming languages include memory transactions in their core, such as Closure, or as a library, such as Java, Haskel, OCaml, Python. In the case of C and C++, there is ongoing work to include it in their standards.

6.2 Challenges

C 6.1. Low-overhead monitoring. A step towards the reconciliation of SMR with the current computer processor architecture, i.e. multicore processors, is to devise new concurrency control protocols that explore pre-ordered transactions to ensure the correctness of a SMR system where individual replicas execute the local operations concurrently [311]. The correctness of such new concurrency protocols must be assessed by intensive testing and monitoring of the system behavior. Any deviations to the specification must be fully diagnosed and corrected. Understanding what is happening at the level of the concurrency protocol itself (including the algorithm internal state and the ordering of concurrent events) plays an important role in such process and must be supported by lightweight (non-intrusive) monitoring techniques, so that the errors are not masked when monitoring is active.

C 6.2. Reduction of the conflicting window. When using the typical API to declare transactions (e.g., begin, read, write, and commit) the system is blind to the application's semantics, i.e., how values read are used by the application. Since transactional speculation is only effective when it succeeds, there is also the need to reduce the number of conflicting transactions by introducing variations in the typical API to declare transactions. The allows clients to express more clearly the intended semantics of the program while executing over an abstract replica state, resulting in fewer conflicts and thus more successful speculative executions. How to reduce both the interactions with the remote database nodes (replicas) and to the "conflicting window" for transactions? Some work has been done on delaying read accesses to the database using futures [40] and double barriers and epochs [279]. Such concepts are still not mainstream in monitoring and logging of transactional systems. Another alternative would be to increase the expressiveness of the transactional API to better express the application semantics and hence improving transactional performance in SMR.

C 6.3. Expressiveness of logs. The performance of concurrency control protocols depends on whether concurrent transactions conflict with each other. The decision of whether two transactions conflict depends on how aware of the concurrency control protocol is of the transactions' semantics. How to do the automatic translation of existing applications into the new transactional SMR infrastructure and how to ensure the new application (using the new transactional API) is functionally equivalent to the original? Any changes to the protocol will create a new transactional infrastructure and any changes to the API will create a new application. In both cases, the new system must be backwards compatible with the original system. Such backward compatibility must be assessed by observing the dynamic behavior of both systems and reason over the collected information to detect any deviations of the new system to the expected behavior. In addition to the huge logs, this challenge raises another question on expressiveness of the logs: What information is registered and how does it express the semantics of the intended transactional operations.

C 6.4. Unification of multiple system huge logs. Observing long living distributed computations such as transactional systems replicated using SMR, may be a main requirement to automatically decompose transactions [330] and/or ensure that the workload is safe [329]. In these cases, if the workload changes or new operations are created, the whole system must be monitored, re-analyzed and re-deployed. In such a distributed setting, possibly many huge logs are collected (one per processor or one per replica) that must be dealt with (see Section 8) and possibly unified into a single log, raising issues on resources' usage and consistency of the multiple observations.

C 6.5. Expressing reparations in transactional systems. In non-transactional applications monitors typically need to have their own reparation code that executes in case the monitor flags a problem. In the case of transactional application monitoring, reparations are readily available and the monitor simply needs to trigger them. While this is more of an opportunity, the challenge lies in how to improve upon current practices and express the behavior of reparations formally and succinctly in a specification language—similarly to the way monitors are defined. There have been several works in this regard [106, 108] for example through the use of *compensating automata*. However, future work can focus on further simplifying the specification language and perhaps providing a library of ready-made constructs which developers can use directly.

C 6.6. Management of historic data to be used in the reparations. From a more pragmatic point of view, compensations and rollbacks present the challenge of managing historic data values to be used in the reparation code. In this respect runtime monitors can be useful in the same way software monitors are typically stateful. Reparations can be parametrized through the monitors' state, avoiding complex wiring to pass the data around. To the best of our knowledge this approach has not been implemented.

C 6.7. Monitoring transactional memory. The time-scale for transactional memory is orders of magnitude smaller than transactional databases. In transactional memory, each access to a shared memory location must be handled by the transactional monitor and considered for the success or failure of the memory transaction. Any additional probing or logging introduced by a monitoring system may influence the scheduling and have a strong impact in a malfunctioning transactional memory application, by changing the serialization order of the transactions, possibly masking or hiding previously observed errors. Researchers have partially addressed this challenge in the past [127, 128, 231, 257] aiming at both correctness and performance.

7 Contracts and Policies

The term *contract* is overloaded in computer science, so it may be understood in different ways depending on the community:

(i) *Conventional contracts* are legally binding documents, establishing the rights and obligations of different signatories, as in traditional, judicial and commercial, activities.

(ii) *Normative documents* are a generalization of the notion of legal contracts. The main feature is the inclusion of certain normative notions such as *obligations*, *permissions*, and *prohibitions*, either directly, or by representing them indirectly. These include legal documents, regulations, terms of services, contractual agreements and workflow descriptions.

(iii) *Electronic contracts* are machine-oriented, and may be written directly in a formal specification language, or translated from a conventional contract. In this context, the signatories of a contract may be objects, agents, web services, etc.

(iv) *Behavioral interfaces* are considered to be contracts between different components specifying the history of interactions between different agents (participants, objects, principals, entities, etc.). Rights and obligations are thus determined by "legal" (sets of) traces which are permissible.

(v) The term "contract" is sometimes used for specifying the interaction between communicating entities (agents, objects, etc.). It is common to talk then about a *contractual protocol*.

(vi) *Programming by contract* or *design by contract* is an influential methodology popularized first in the context of the programming language Eiffel [239]. "Contract" here means a relation between pre- and post-conditions of routines, method calls, etc. This concept of contract is also used in approaches such as the KeY program verification tool [212].

(vii) In the context of web services, "contracts" may be understood as *service-level agreements* usually written in an XML-like language like IBM's Web Service Level Agreement (WSLA [325]).

(viii) More recently, the term "contract" is used in the context of *blockchain* and other *distributed ledger technologies* as programs that ensure certain properties concerning transactions. These programs are called *smart contracts* [305], as popularized by the Ethereum platform [87].

In this section we focus on the use of the term in the computational domain but with a richer interpretation than just a specification or property. In particular, we consider two types of contracts: (ii) normative documents (including conventional contracts and their electronic versions as described above), and (viii) smart contracts. In both cases, we refer to *"full contracts"* [258], that is agreements between different entities regulating not only the normal interactive behaviors, but also exceptional ones. A common aspect of such contracts is that they should express not only the sequence and causality of events, but also what obligations, permissions and prohibitions the participating entities have (basic modalities studied in deontic logic [324]), as well as the associated penalties in case of violations.

An example of a full contract in the case of a normative document in the context of a stringent renting agreement, would be one containing for instance the following clauses (among others): *"1. The tenant must pay 200 EUR, in advance, on the 5th of each calendar month. 2. In case of not complying with clause 1, the tenant will have till the 15th of the month to pay the above mentioned sum plus an additional fee of 5% of the amount. 3. In case of not complying with clause 2, the tenant will have to leave the premises before the end of the month and the deposit will be retained by the landlord."* Note that the contract includes clauses which may be violated, but includes reparatory clauses to cover such cases. Although violating clause 1 and paying late is a behavior covered by the contract, it is clearly less desirable (in terms of compliance) than if clause 1 were to be satisfied. In the case of a smart contract, the corresponding program should implement all the above, including the exceptional behavor (i.e., not only the primary obligations but also enforce the penalties associated with the non-compliance of such obligations). A contract not containing clauses stipulating the penalties and deadlines associated with the non-compliance with the written obligations, would not be considered to be a *"full contract."*

The specification of such contracts requires a formal language rich enough to capture these deontic notions, temporal and dynamic aspects, real-time issues such as deadlines, the handling of actions (events) and exception mechanisms. The main aim is not only to specify such contracts, but to analyze them using techniques like model checking and runtime verification. Clearly, the use of contracts is only meaningful if there is a mechanism to validate their fulfillment.

A related concept is that of *policies*. At a certain level of abstraction, policies can be seen as contracts in the sense that they prescribe behavior. Since the term policy is also very generic with a broad scope, we concentrate on *privacy policies* (or privacy settings) and more specifically in the context of Online Social Networks (OSN) like Facebook and Twitter.

As mentioned before, deontic logic is a natural formalism to represent normative documents as they mostly talk about obligations, permissions and prohibitions, as well as to capture what happens in case of violations. In the case of privacy policies, one may be interested in prescribing who should *know* what about whom and under which circumstances. So, it makes sense then to use *epistemic* logic [140] to reason about privacy policies. That said, note when describing such policies we informally use deontic modalities, who *should* (not) access certain information, and who is *allowed* to perform certain actions (e.g., to make a friend request). Those (deontic) normative concepts are, however, not needed as primitives in this context. Giving a detailed explanation on why this is the case is beyond the scope of the paper (see for instance the formalization of privacy policies for OSNs presented in [263,261,262]).

What is important here is that from a runtime verification perspective, monitoring privacy policies for OSNs and normative documents, have similarities mostly in what concerns their challenges as explained at the end of this section.

7.1 Context and Areas of Interest

We provide now some more detail context of the following aspects of contracts: contracts as normative documents, the so-called smart contracts, and policies for online social networks.

7.1.1 Contracts: Normative documents

The complete specification of full contracts —normative texts which include tolerated exception, and which enable reasoning about the contracts themselves— can be achieved using a combination of temporal and deontic concepts [258]. Formalizing such contracts requires operators and combinators for choice, obligations over sequences, contrary-to-duty obligations, and the representation of how internal and external decisions may be incorporated in an action- or state-based language for specifying contracts. There have been several interpretations and approaches for the development of such a logic [258], including modal extensions of logics and automata in order to address the issue of how contracts can be formalized and reasoned about. See, for example [37,88, 168,229,243,273,274,326], just to mention a few.[6]

Why is there a need for a logic or some other formal language? One of the aims of formalizing contracts is not simply to use them as specification, but also to be able to prove properties about the contracts themselves, to perform queries on the contracts (like what each party is agreeing to), and ultimately to ensure at runtime that the contract is satisfied (or alternatively to detect for violations). An alternative approach is to use *machine learning* (or other artificial intelligence techniques). For instance, one may avoid the use of formal methods by using *natural language processing* (NLP) combined with machine learning to directly perform queries on the textual representation. While this is feasible in certain cases, it is well known that the state of the art in NLP is still far from being able to deliver fully automatic and sufficiently reliable techniques. Moreover, performing semantic queries or running simulations still require a formal representation. This is an important and interesting research area in itself, but here we are concerned not with the problems of obtaining such normative documents but with the specific issue of monitoring their satisfaction or violation.

In terms of monitoring of contracts, most of the current work start from some form of formal semantics. There are various outstanding questions of what subsets of deontic logics are tractably and practically monitorable. For example, are more standard logics, like classic or temporal logics, enough? How important is to get full complex semantics (e.g., based on Kripke semantics) for the logic? For a full representation and analysis of contracts, Kripke semantics might be necessary, but for monitoring purposes a much simpler approach considering trace semantics seems to be sufficient.

Concerning monitoring, an ideal goal is to automatically extract a monitor from the document's formal representation, but this is, in general, not feasible. We assume then that we obtain the monitors from a given contract manually or semi-automatically. This is still not an easy task, as there is no standard, easy and direct way to extract a model from a document in natural language.

The use of *controlled natural languages* (CNL) [216] has been proposed in different works in order to facilitate bridging the gap between the natural language description of the original document and a more formal representation in the form of a formal language [88,90,327]. In a legal specification setting, there is initial work in this direction, but we are still far from reaching this goal [85,89,90].

7.1.2 Smart contracts

If the computer science community borrowed the notion of contracts by remarking on the similarity between specifications and legal agreements, the legal community saw an opportunity in viewing computer code as a form of executable enforcement or enactment of agreements or legislation. The notion that executable code regulates the behavior of different parties very much in the same manner that legal code does was proposed by Lessig [224] in 1999. The dual view, that the use of executable smart contracts can enforce compliance as an integral part of the behavior, was argued earlier by Szabo in 1996 [305].

The introduction of blockchain [248] and other distributed ledgers technologies, which enable the automated management of digital assets, has changed the way in

[6] The literature is quite vast and the list of citations is not exhaustive. The main conferences, workshops and journals in the area include JURIX [205], DEON [124], RuleML [288], and the Journal of Artificial Intelligence and Law [199].

which computer systems can regulate the interaction between real-world parties. In particular, these technologies have enabled the deployment of Szabo's notion of smart contracts in a distributed setting, without the participation of trusted central authorities or resource managers. For instance, the Ethereum [322] blockchain supports smart contracts which can be expressed using a Turing-complete programming model, to be executed on the Ethereum Virtual Machine (EVM) and typically programmed using one of a number of languages supporting a higher level of abstraction.

Smart contracts are executable specifications of the way the contract will update the state of the underlying system. Although specifications can be executable or not (see [166] and [185]), it is generally accepted that executable specifications must elucidate *how* to achieve the desired state of affairs, while non-executable specifications simply characterize properties that the desired state should satisfy. The former is substantially more complex, which is why the fields of validation and verification arose to explore ways in which executable specifications (code) can be verified against non-executable ones (properties).

This gives rise to a challenge: that of verifying that smart contracts indeed perform as they should. Although one can argue that the challenge behind verification of such executable code is no different from that of verifying standard programs, there are a number of issues which are particular to smart contracts. There has been little work yet addressing the special idiosyncrasies of smart contracts. Static analysis techniques for the verification of smart contracts has been proposed in [76], via a translation from smart contracts into another language (F* in this case) for verification. See [10] for a discussion on some challenges concerning the verification of smart contracts using deductive verification techniques. From a runtime perspective, there has been some work on using blockchain technology to regulate distributed systems (see [170,178, 275,317]), but the focus of this work is not on the verification of the smart contracts themselves. Initial attempts to address runtime verification of smart contracts and building tools to automate this have started to appear [103,138], but many challenges remain to be addressed [36].

One particular aspect that presents specific challenges is that these smart contracts are typically mainly concerned with the movement of digital assets, with built-in notions of failing transactions and computation roll-back to handle failure. Although this has been investigated in the domain of financial system verification [108,264], there is a major difference. Before the rise of cryptocurrencies, all such systems were deployed on a central trusted system, typically residing within the infrastructure of the payment institution. In contrast, in the context of distributed ledgers, the storage and computation are, by their very nature, distributed, and particularly runtime verification require the instrumentation and deployment to take this into consideration.

There is a major difference with regular financial transaction software deployed on, or interacting with, payment institutions. That is that given the critical nature of such systems (payment applications have been built using a strict validation process) ensuring compliance to legislation and adherence to specifications. However, with what has been hailed as the democratization of currency systems, came the popularization of payment application development, with many smart contracts being developed without the necessary care and responsibility. This approach has suffered a number of huge financial losses due to bugs [33]. The need for lightweight runtime validation of such systems, whether inbuilt in the execution of the smart contracts or inherent in the blockchain or alternative distributed ledger technology is essential to ensure user safety.

Turing-complete environments for smart contracts suffer from the possibility of non-termination or excessively long computation. Rather than limit the power of the programming language, the solution adopted in systems such as Ethereum was that of introducing the notion of *gas* —a resource required to enable computation and that has to be paid for using other digital assets, typically the underlying cryptocurrency. Although efficiency of computation has always been an important issue in computing, it has typically been detached from functional correctness issues addressed by formal methods. With the notion of gas, the direct correlation between execution steps and financial cost is a new challenge for runtime verification. As a direct corollary, additional computation to check for correctness will directly induce additional cost. However, there is also the issue that gas affects computation, in that once gas runs out, computation is reverted, which has been exploited in a number of smart contract attacks. Finally, the use of gas throughout the computation may justify qualitative dynamic analysis to measure the extent of satisfaction or violation using a distance metric to detect failure due to lack of gas.

Finally, the multitude of contracts and interaction platforms provided by the underlying distributed technology is likely to give increased importance to contract comparison and negotiation. We envision a scenario, in which one may negotiate for increased dependability (e.g. by monitoring additional logic) against a stake paid by the developer or provider of the contract. At a more complex level, one can have a system where different or additional functionalities are negotiated upon setting up a smart contract. In both cases, the process is a form of meta-contract which regulates how the parties may interact to negotiate and agree upon a contract which will be set up.

See [10] and references therein for a discussion on the verification of smart contracts, as well as papers in [196] for recent advances and a discussion on open issues in the area.

7.1.3 Privacy policies for OSNs

Policies may be understood, at a certain level of abstraction, as contracts: they prescribe what actions are allowed or not. The term policy is generic and may be applied to many different cases or applications. We focus here on privacy policies, and in particular on privacy policies for Online Social Networks (OSNs). OSNs provide an opportunity for interaction between people in different ways depending on the kind of relationship that links them. One of the aims of OSNs is to be flexible in the way one shares information, being as permissive as possible in how people communicate and disseminate information. While preserving the spirit of OSNs, users would like to be sure that their privacy is not compromised. One way to do so is by providing users with means to define privacy policies and provide them with guarantees that their requested policy will be respected.

For defining policies one might use simple checkbox privacy settings (as it is the case in most OSNs today), or allow user to define more richer policies using expressive formal languages or logics. Given means to specify privacy policies is not enough, as these policies must be enforced at runtime. Enforcement of checkbox privacy settings is rather well-understood, at least for most of the kind of policies currently implemented in existing OSNs. However, if one wants to allow the definition of richer policy languages, the challenge goes beyond identifying an appropriately expressible language to the problem of automatically extracting a runtime monitor to act as an enforcement mechanism. This is currently beyond the state of the art and no concrete solutions exist.

Furthermore, the state of the art today is focused on static policies. For instance, in Facebook users can state polices like *"Only my friends can see a post on my timeline"* or *"Whenever I am tagged, the picture should not be shown on my timeline unless I approve it"*. However, no current OSN provides the possibility of defining and enforcing *evolving* (dynamic) privacy policies. Policies may evolve due to explicit changes done by the users (e.g., a user may change the audience of an intended post to make it more restrictive), or because the privacy policy is dynamic per se. Consider for instance: *"Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home"*, *"Only up to 3 posts disclosing my location are allowed per day on my timeline"*, *"My boss cannot know my location between 20:00-23:59 every day"*, and *"Only my friends can know my location from Friday at 20:00 till Monday at 08:00"*. No current OSN addresses the specification and enforcement of such policies. Formal languages are needed to express such time and event-dependent recurrent policies, and suitable enforcement mechanisms need to be defined. This could be done by defining real-time extensions of epistemic logic, or combining existing static privacy policy languages with automata, as done for instance in [260,261,262].

7.2 Challenges

C 7.1. Formalizing natural language contracts. A major challenge is the identification of techniques to extract a formal model from a normative document in an automatic manner. In particular, the challenge is to adapt NLP techniques and use machine learning techniques to (semi-)automatically translate natural language text into a suitable CNL.

C 7.2. Formal reasoning about legal documents. A challenge in the formalization of legal documents is the choice of the right formal language adequate for the type of analysis required, as there is a trade-off between expressiveness and tractability. In particular, the notion of *permission* (and *rights*) poses challenges in monitoring, since one party's permission to perform an action typically entails an obligation on the other party to allow the action, and this obligation may not be observable unless the right is exercised.

C 7.3. Operationalization of legal documents. Most legal texts are written in a declarative style, and typically require to be operationalized for automated analysis. Furthermore, parts of these texts may refer to events or attributes which are not observable and thus not monitorable. Most runtime monitoring and verification approaches for legal texts interpret the term *runtime* to refer to the time during which the legal text regulates. Another possible interpretation is that of monitoring the process of drafting of a contract or legislation, or the negotiation of a contract. A monitoring regime could be useful in this setting.

C 7.4. Smart contract monitoring and verification. How to adapt dynamic verification to smart contract monitoring is unclear, particularly because once a problem arises, it is not always possible to take reparatory action to recover. An open question is how enforcement, verification and reparation can be combined in a single formalism and framework.

C 7.5. Monitoring gas in smart contracts. Another challenge is the use of the notion of 'gas' to justify computation on ledger systems such as Ethereum, although it is unclear how dynamic analysis can be used effectively to track such a non-functional property. Furthermore, the introduction of runtime verification overheads in terms of gas poses new challenges for monitoring.

C 7.6. Compliance between legal and smart contracts. The relation between the underlying legal document and smart contracts is still to be addressed. The challenge here is how to monitor compliance between both versions of the contract, and relate violations in the execution of the smart contract with the corresponding clause in the real legal contract.

C 7.7. Policy monitoring and verification. The challenges we identified for contracts also apply to policies. In particular, there might be a need to combine the enforcement mechanism with machine learning techniques and with natural language processing. For instance, a post might contain a sentence like *"I am here with John drinking a glass of wine"*, where *"here"* clearly refers to a place which might be inferred from the location associated with the post. This kind of inference is difficult to do automatically by machine.

C 7.8. Policy monitoring in OSNs. For Online Social Networks (OSNs), the use of epistemic logic to reason about whether and how explicit (and derived) knowledge of users adhere to policies has been explored. However, the operationalization of such policies and the extraction of monitors from policies have proved to be particularly difficult.

C 7.9. Policy monitoring and verification. The evolution of policies due to specific events or timeouts also poses a number of challenges. Some initial work has been recently done on the specification side with a proof of concept implementation. The work in [261,262] presents an approach based on extending a privacy language with real-time, while [260] proposes a combination of static privacy policy language with automata. However, a general working solution to this challenge is still missing.

8 Huge Data and Approximate Monitoring

This section describes runtime verification challenges related to the analysis of very large logs or streams of events from the system under observation. The general goal when dealing with huge data streams is to develop algorithms that offer scalability, specification language expressiveness, accuracy, and utility. Below we discuss the advances made along each of these dimensions and some of the remaining challenges.

8.1 Context and Areas of Interest

Before we present the challenges for RV in the area of huge data and approximate monitoring, we first provide some context and state-of-the-art related to the following areas: scalability, expressiveness, accuracy and utility.

8.1.1 Scalability

In runtime verification, the focus to date has mainly been on efficiency, expressiveness, and correctness, and less so on scalability to *Big Data* in realistic scenarios. A few exceptions exist and are summarized below, which mostly address offline monitoring.

Barre et al. [42] and Hallé and Soucy-Boivin [183] use Hadoop's MapReduce framework to scale up the monitoring of propositional LTL properties using parallelization. In their experiments, they used event logs with more than nine million entries. In these approaches, formulas are processed bottom up using multiple MapReduce iterations. While the evaluation in the map phase is completely parallelized for different time points from the event log, the results of the map phase for a subformula for the whole log are collected and processed by a single reducer. In a single iteration there are as many reducers as there are independent subformulas with the same height. The reducers, therefore, become bottlenecks that limit the scalability.

Bianculli et al. [77] extend this approach to the offline monitoring of large traces, for properties expressed in MTL with aggregation operators. Similarly to the aforementioned approaches, the memory consumption of the reducers limits the scalability of this approach. More specifically, reducers (that implement the semantics of temporal and aggregate operators) need to keep track of the positions relevant to the time window specified in the formula: the more time points there are the denser the time window becomes, with a consequent increase in memory usage. Bersani et al. [72] worked around this problem by considering an alternative semantics for MTL, called the *lazy semantics*. This semantics evaluates temporal formulas and Boolean combinations of temporal-only formulas at any arbitrary time instant. It is more expressive than the point-based semantics and supports the sound rewriting of any MTL formula into an equivalent one with smaller, bounded time intervals. The lazy semantics has the drawback that basic logical properties do not hold anymore. This disallows formula simplifications and complicates the formalization of properties given in natural language, since familiar concepts have a different meaning. Unlike the previous approaches, Bersani et al. implemented the monitor on top of the Apache Spark framework [337] that is optimized for iterative distributed computations.

Parametric trace slicing [100,280] is a technique for monitoring a parametric LTL property by grounding it to several plain LTL properties. In this approach logged events are grouped into slices based on the values of the parameters. A slice is created for each parameter value or for each combination of values depending on the number of parameters. The individual slices are then processed by a propositional LTL monitor unaware of the parameters. The initial main goal of this approach was not scalability, but rather monitoring the more expressive parametric LTL specification language. However, the approach is also relevant for scalability since it easily lends itself to parallelization.

Another line of work [58,53] similarly splits the logged events into slices, but it avoids grounding first-order properties altogether. This is enabled by using a more powerful monitor, MonPoly [55,59,61,62], to process the slices. Overall, the approach allows for scalable offline monitoring of properties expressed in *Metric First-Order Temporal Logic* (MFOTL). The core idea in this work is to split the log into multiple slices and check the same formula on each slice independently. This allows the solution to scale, by handling one slice on a single computer. The key component is a log-splitting framework used to distribute the log to different parallel monitors based on data and time. The framework takes as input the formula and a splitting strategy and splits the log ensuring soundness and completeness. The approach was implemented in Google's MapReduce framework where the log-splitting framework is executed in the map phase. The approach is, however, limited to offline monitoring since it uses MapReduce. Parallelization is not limited as in the previous approaches, but it is potentially wasted, since to ensure correctness, the log splitting framework may completely duplicate the original log into some of the individual slices. Another limitation is that the slicing framework relies on a domain expert to supply a splitting strategy manually. For example, if a monitored property involves events parametrized with "servers" and "clients", one could split the log along the different "servers", along the different "clients", or along both.

Loreti et al. [230] discuss two MapReduce architectures to tame scalability in the context of compliance monitoring of business processes, using the SCIFF framework [13]. Such a framework provides a logic-based proof procedure for checking declarative constraints on sequences of events, in terms of expectations and happened events. The two MapReduce architectures proposed in this work were adapted from similar ideas in process mining [312] and distinguish between *vertical* and *horizontal* distribution. In the vertical distribution all nodes receive the complete specification and a subset of the complete log. During the map phase, the log is split across the various nodes such that all the events of a trace are sent to the same node. In the reduce phase, each node checks the conformance of each log fragment to the specification. In horizontal distribution both the specification and the logs are partitioned across the nodes. Each node checks a partial specification on a fragment of the log that contains only the events used in the partial specification. The results of all the nodes are then merged together with a logical AND. The limitation of the approach is the expressiveness of the SCIFF logic programming framework that cannot handle parametric specification.

Yu et al. [336] propose an approach for parallel runtime verification of programs written in the *Modeling, Simulation and Verification Language* (MSVL), with properties expressed in *Propositional Projection Temporal Logic* (PPTL). The approach divides each program trace into several segments, which are verified in parallel by threads running on under-utilized CPU cores. The verification results of all segments are then merged and further analyzed to produce a verdict.

8.1.2 Expressiveness

Most of the works on runtime verification borrow logics from static verification approaches and focus on designing algorithms that either (1) generate a monitor that can analyze a trace online, or (2) can process dumps of traces offline. Optionally, one could use a general programming language or a domain-specific language to write the queries that process the input traces online or offline. In both cases, we would like to monitor Big Data with a highly expressive specification language. More expressive logics naturally require more computation resources for monitoring. Thus, a worthwhile research question is: *What are the limits of the specification language expressiveness to achieve scalable monitoring of Big Data?* Below we discuss some directions of how expressive specification languages could look like.

Complex Event Processing (CEP) and *Data Stream Management Systems* (DSMS), for example, can serve as specialized languages for building stream processors (see [238] for a recent survey). The query languages of DSMS are mostly extensions of SQL (e.g., with window operators [21]), and thus typically much weaker than logics such as MFOTL due to the absence of proper negation and more limited capabilities for expressing temporal relationships. Moreover, DSMS tend to focus on efficient query execution at the expense of sacrificing a clean semantics of the property specifications. The reference model of DSMS has been defined in the seminal work on the *Continuous Query Language* (CQL) [21]. In CQL, the processing of streams is split in three steps. i) Stream-to-relation operators—that is, windows—select a portion of each stream thus implicitly creating static database table. ii) The actual computation takes place on these tables, using relation-to-relation (mostly SQL) operators. iii) Finally, relation-to-stream operators generate new streams from tables, after data manipulation. Several variants and extensions have been proposed, but they all rely on the same general processing abstractions defined above.

CEP [232, 238] systems are closely related to DSMS. CEP systems analyze timestamped data streams by recognizing composite events consisting of multiple atomic events from the original stream that adhere to certain patterns. The user of a CEP system controls the analysis by specifying such patterns of interest. The predominant specification languages for patterns are descendants of SQL [181]. An alternative is given by rule-based languages, such as Etalis [16], which resembles Prolog. Although CEP systems improve the ease of specification of temporal relationships between events over DSMS, they are still significantly less expressive than MFOTL due to their restricted support for parametrization of events and lack of quantification over parameters. Interestingly, some CEP systems use interval timestamps. In this model, each data element is associated with two points in time that define the first and the last moment in time in which the data element is valid [296, 319].

For logical specification languages such as LTL a recent trend has been to incorporate regular-expression-like constructs in the logic. This gave rise to the industrially standardized *Property Specification Language* (PSL) [135], the development of *Regular Linear Temporal Logic* (RLTL) [225, 293] and its more recent incarnation in the form of *(Parametric) Linear Dynamic Logic* ((P)LDL) [173, 148] and its metric counterpart (MDL) [56]. Due to the extension with regular expressions, those languages are more expressive than LTL in that they capture all ω-regular languages. Vardi [313] observed that these extensions were essential for the practical usage of PSL in many industrial application settings. First-order extensions of languages like PSL, RLTL, (P)LDL, and MDL, which should be more expressive than MFOTL, have not yet been considered for monitoring.

However, to keep things manageable for Big Data, it may be necessary to restrict or even remove features from our property specification languages. The usage of negation is a candidate for restriction while the first-order aspect of MFOTL is a candidate for removal (or for replacement with freeze quantifiers). Many works [53, 60, 61, 62, 64] had to define (efficiently) monitorable fragments using similar restrictions. A syntactic restriction (e.g., of the allowed occurrences of negation) is preferable over a modification of the semantics as seen on the example of negation in many data stream management systems (DSMS). The user of a specification language with a syntactic restriction can at least rely on the familiar semantics. Moreover, properties outside of the monitorable fragment can be often automatically rewritten into equivalent formulas within the fragment.

8.1.3 Accuracy

Compromising on soundness is not a common approach in runtime verification. However, when faced with very large logs (or streams) of data and hard real-time constraints on providing verdicts, it can become a very useful compromise. In some cases, sound algorithms cannot be used in practice. For example, a sound algorithm that determines the number of distinct elements in a data stream must use space linear in the cardinality it estimates, which is often impractical. Determining cardinality is a large component of many practical monitoring tasks such as detecting worm propagations, denial of service (DoS) attacks, or link-based spam. Ideally, tradeoffs between monitoring efficiency and accuracy of the provided verdicts should be formulated as an additional input to the monitor. We call such an extension *approximate monitoring.*

Approximate monitoring deals with the issue of providing approximate (or inaccurate) results to the standard monitoring problem, with bounds on the "distance" between the actual (correct) results and the provided ones. The definition of such a distance depends on the particular output that a monitor provides. For instance, in the case of a simple stream of violations, distance can be defined as the percentage of unreported violations, or the percentage of spuriously reported violations. For other monitoring outputs that contain richer verdicts, distance can be defined to further include the accuracy of the additional information in the verdicts.

One should make a clear distinction between approximate monitoring and monitoring probabilistic properties. The latter deals with monitoring specification languages that can express probabilistic and statistical properties of data streams. However, it still provides correct verdicts given the semantics of the specification language. A related facet is the monitoring of uncertain data, which deals with the problems of data collection and data reliability, and it often carries over to monitoring by invalidating certain assumptions on the data stream. There are many sources of uncertainties in the monitored data: timestamps can be imprecise due to clock skew, logs may be incomplete due to outages, or even disagree when coming from various sources. Uncertainty can come from the monitored systems themselves which can exhibit stochastic and faulty behavior. Another related field is state inference of the monitored system using probabilistic approaches where a belief state is maintained and updated during monitoring. Although these approaches provide probabilistic guarantees as part of the resulting belief state, they perform a specific monitoring task.

Existing work on approximate monitoring stems from the fields of databases [38], streaming algorithms [251], and property testing [176]. All approaches can be classified based on two criteria: the specific queries they approximate and the resources they optimize. Commonly approximated queries in the literature are cardinality estimation [152], top-k items [39], frequent items (heavy hitters) [211, 237, 335], quantiles [113, 335], frequency moments [111, 114], entropy [20], other non-linear functions over (possibly distributed) streams, and distance queries [9]. Orthogonally, the approaches either optimize memory consumption, communication cost, execution time, or the monitor's overhead.

Optimizing memory consumption has led Morris to develop his well-known approximate algorithm for counting [244]. The HyperLogLog algorithm [152] tackles the cardinality estimation problem mentioned in the example above. Counting the most frequent items in a stream is a very common query. In fact there has been an ample amount of work in devising good approximation algorithms. One of the oldest streaming algorithms for detecting frequent items is the MJRTY algorithm [82] and its generalizations [121, 209, 241].

Optimizing communication cost is a common problem in the field of streaming databases. Consider k data streams and a monitor that consisting of $k + 1$ distributed components —one for every stream and an additional central coordinator. Components are only allowed to send messages to the central coordinator. The goal is to track a (reasonably accurate) value of a function defined over the data in all k streams at the central coordinator, while minimizing the number of messages sent. This problem is a good abstraction of many network monitoring tasks where the goal is to detect global properties of routed data. The communication cost is the primary measure of complexity of a tracking algorithm. Initial work dealt with optimizing the top-k items query [39]; it was then extended to non-temporal functions [114, 323]. Temporal queries are facilitated by introducing various types of windows, and the approximation is achieved by maintaining a uniform sample of events per window at the coordinator [110, 115].

Optimizing execution time using approximation methods involves ignoring parts of the input, predicated on strong statistical guarantees on the accuracy of the output. This is enabled by sampling techniques [112] that are shown to work for specific queries. These techniques are often referred to as Approximate Query Processing (AQP) and they are implemented by many existing systems [9, 245, 246, 302]. When sampling, a random sample is a "representative" subset of the data, obtained via some stochastic mechanism. Samples are quicker to obtain, smaller than the data itself and are hence used to answer queries more efficiently. A histogram summarizes the data by grouping its values into subsets (or "buckets") and then computing a small set of summary statistics for each bucket. These statistics allow to approximately reconstruct the data in each bucket. Wavelets are techniques by which a dataset is viewed as a set of M elements in a vector, i.e., a function defined on the set $\{0, 1, 2, \ldots, M-1\}$. Such a function can be seen as a weighted sum of some carefully chosen wavelet "basis functions". Coefficients that are close to zero in magnitude can then be ignored, with the remaining small set of coefficients serving as the data summary. Sketches are particularly well-suited to streaming data. Linear sketches view a numerical dataset as a matrix, and multiply the data by some fixed matrix. Such sketches are massively parallelizable and used to successfully estimate answers to set cardinality, union and sum queries, as well as top-k or min-k queries.

Optimizing monitoring overhead is a problem often encountered in runtime verification. When optimizing overhead, one must consider the monitored system in addition to the event stream. In this setting, computing resources (time, memory, and network) are shared by the monitored system and the monitor. Overhead can be seen as the percentage of the resources used by the monitor. Bartocci at al. [50, 206, 303] use dynamic knowledge about the monitored system to control the amount of resources that are allocated for monitoring. More precisely they enable and disable monitoring of certain events as needed. This can be seen as sampling, however the stochastic mechanism is informed by the probabilistic model of the monitored system. Given how likely it is that an event will participate in a violation of a given temporal property, the system decides to include it in the monitored stream. The aforementioned approaches all differ in the probabilistic formalism used to model the monitored system [49].

8.1.4 Utility

Another important dimension is the usefulness (or utility) of the monitoring output. The expected output of the monitoring problem is often underspecified and usually different approaches employ different assumptions derived from the implementation details of the monitoring algorithms. Yet, the underlying time and space complexity of the monitoring problem highly depends on its precise output specification.

For instance, some monitoring algorithms output a single Boolean verdict stating that, overall, the trace satisfies or violates the monitored property. Other monitoring algorithms solve a strictly harder problem - they output a stream of Boolean verdicts attesting to the satisfaction of the monitored property for every prefix of the trace (or stream). While the complexity of the former variants have been studied for various specification languages [84, 149, 217], the latter have mostly been ignored.

An interesting distinction to make is between outputting a stream composed only of violations, *versus* giving a (more general) stream of verdicts that includes satisfactions of the monitored property as well.

Traditional monitoring algorithms for temporal logics with future operators, scale poorly when subjected to high-velocity event streams. One reason is that the monitor is constrained to produce outputs strictly in the order defined by the incoming events. It can be shown that this ordering constraint, although providing more usable output, makes for a more complex monitoring problem. An interesting special case of monitors producing out-of-order output are monitors that output violations as soon as possible, i.e., as soon as they have enough information from the input to pinpoint some violation. Monitors that produce ordered output violate this seemingly natural monitoring requirement.

Orthogonally, in contrast to reporting all violations of a property, there are many valid use cases where monitors report only some (most relevant) violations. Examples include reporting only the first, or the last (most recent) violation. However, the impact of these choices on the monitoring complexity is unclear.

It is also possible to design algorithms that produce non-Boolean verdicts, for example using *Stream Runtime Verification* [120], which allows to compute streams from arbitrary domains. Other system use verdicts that target specific (potentially relaxed) output requirements and may or may not contain enough information to reconstruct the standard Boolean verdict output. For example, Basin et al. [57] proposed the so-called equivalence verdicts that state that the monitor does not know the Boolean verdict at a particular point in the event stream, but it knows that the verdict will be equal to another, also presently unknown, verdict at a different point. The equivalence verdicts carry enough information to reconstruct a stream of Boolean verdicts. To do so, one must reorder the verdicts reported in the output stream and propagate Boolean verdicts to the equivalent ones.

All output variations mentioned so far compromise utility for the sake of scalability. However, sometimes starting from a stream of verdicts, it is quite nontrivial to understand why a complex property is satisfied (or violated) at some point in the trace. One can increase the utility of the monitors by replacing the stream of Boolean verdicts with a stream of proof objects that encode the explanations as to why property has been satisfied or violated. The proof objects can take the forms of minimal-size proof trees [52], or a compressed summary trace capturing the essentials of the original trace that contribute to a violation.

8.2 Challenges

C 8.1. Combining Horizontal and Vertical Parallelization. The different approaches to parallelize monitoring algorithms have different advantages and limitations. Horizontal parallelization as in Barre et al. [42] and Hallé and Soucy-Boivin [183] does not dependent on the actual events but is limited by the formula's structure. Vertical parallelization as in Basin et al. [53] or parametric trace slicing [100, 280] offers an *a* *priori* unbounded amount of parallelization but may lead to data duplication for certain formulas. A combination of the approaches may achieve the best of both worlds and is worth investigating.

C 8.2. Scalable Monitoring in Online Setting. Most of the described approaches rely on MapReduce as a technical solution for distributed fault tolerant computation. However, its batch-processing nature restricts monitoring to the offline setting, in which the complete log of events is given as input to the monitor at once. More recently, systems research has moved towards a proper streaming paradigm, as witnessed by widely adopted streaming frameworks such as Apache Flink [91] or Timely Dataflow [247]. These frameworks can be used to achieve scalability in the online setting, in which individual events steadily arrive at the monitor. The challenge thereby is to adapt the offline approaches (both horizontal and vertical) to the online setting.

C 8.3. Adaptive Scalability. A related challenge that arises only in the online setting is adaptivity. To retain scalability, a parallel monitor, and in particular its log slicing component, may need to adapt to changes in behavior of the monitored system. For example, an event-rate increase or change in the occurrence distribution of some system events. Detecting such changes and adequately reacting to them are both challenging. In particular, the latter will most likely require a reshuffling of the parallel monitors' states in a way that maintains a consistent global state, that is, it does not compromise the soundness of monitoring.

C 8.4. Automatically Synthesizing Splitting Strategies. Log slicing techniques, like Basin et al. [53] rely on a domain expert to supply a splitting strategy. An open challenge is how to synthesize such a splitting strategy automatically, based on the monitored property and some formalized domain knowledge, for example, statistics on types of events in the log. The holy grail would be an algorithm that picks the *optimal* splitting strategy, i.e., one that minimizes the amount of duplicated data between the slices and creates balanced slices that require equal computational effort to monitor.

C 8.5. Expressive Specification Languages. Richer specification languages allow to capture more sophisticated properties. For example, hyperproperties allow to express relational properties (essentially properties that relate different traces). These traces can come from a single large trace that is processed offline. For example, a specification can relate two traces, which are extracted from the large trace as requests coming from different users or different requests performed at different points in time. This richer language would allow to express properties like differential SLA that are beyond the expressiveness of the specification formalisms currently used. Another family of specification languages that allow to express rich properties is stream runtime verification languages. Currently, these languages only have online and offline evaluation algorithms for small traces, in the sense of traces that can be stored in a single computer. A challenge is then to come up with parallel algorithms for large traces.

C 8.6. Richer Verdicts and Concise Model Witnesses. Classical specification formalisms from runtime verification, borrowed from behavioral languages used in static verification, generate Boolean outcomes from a given trace, which indicate whether the trace observed is a model of the specification. One challenge is to compute richer outcomes of the monitoring process. Examples include computing quantitative verdicts, like for example how robustly was the specification satisfied or computing statistics from the input trace, like the average number of retransmissions or the worst-case response time. A related challenge is the computation of witnesses of the satisfaction or violation of the property for offline traces. The main goal is that the monitoring algorithm computes the verdict and, as by-product, a compressed summary trace, where irrelevant information has been omitted and consolidated. Algorithms will have to be created to (1) check that the summary trace is indeed a summary of the input trace, and (2) that the summary trace has the claimed verdict against the specification. This process, if successful, will allow to check fast and independently that the runtime verification process was correctly performed.

C 8.7. Approximate monitoring. The monitoring setting should provide a systematic and explicit way to specify tradeoffs between the resources the monitoring algorithms may utilize (e.g., maximum memory consumption or running time) and the accuracy of the verdicts they provide. Existing work provides such tradeoffs for a few fixed monitored properties (usually involving aggregations), however, support for complete language fragments is an open problem.

C 8.8. Impact of utility on monitoring complexity. The existing work on the complexity of monitoring [84, 149, 217] (called path checking in this context) only considers the problem of providing a single Boolean verdict in an offline manner. Tight complexity bounds for the online monitoring problem or other variants of the problem with different output utility (e.g., a verdict stream) have not yet been established. The impact of the different kinds of verdicts on the complexity of the resulting monitoring problem needs to be better understood.

9 Conclusion

Runtime verification techniques have been traditionally applied to software in order to monitor programs. One of the missions of the EU COST Action IC1402 (*Runtime Verification beyond Monitoring*) was to identify application domains where runtime verification and monitoring could be applied, and describe the challenges that these domains would entail. This paper has explored seven selected areas of application, namely, distributed systems, hybrid systems, hardware based monitoring, security and privacy, transactional systems, contracts and policies and monitoring large and unreliable traces. For each of these seven domains, we survey the state-of-the-art focusing on monitoring techniques in these areas, and finally presented some of the most important challenges (collecting a total of 47 challenges) to be addressed by the runtime verification research community in the next years.

Acknowledgements

This research has been supported by the European ICT COST Action IC1402 *Runtime Verification beyond Monitoring (ARVI)*. The authors would like to thank Fonenantsoa Maurica and Pablo Picazo-Sanchez for their feedback on parts of a preliminary version of this document, and the anonymous reviewers for the constructive corrections, comments and criticism.

References

1. M. Abadi and D. G. Andersen. Learning to protect communications with adversarial neural cryptography. Technical Report CoRR abs/1610.06918, arXiv.org, 2016.
2. H. Abbas, G. E. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(s2):95:1–95:30, 2013.
3. H. Abbas, A. Winn, G. E. Fainekos, and A. A. Julius. Functional gradient descent method for metric temporal logic specifications. In *Proc. of the American Control Conference (ACC'14)*, pages 2312–2317. IEEE, 2014.
4. L. Aceto, A. Achilleos, A. Francalanza, and A. Ingólfsdóttir. Monitoring for silent actions. In *Proc. of the 37th IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science, (FSTTCS'17)*, volume 93 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
5. L. Aceto, A. Achilleos, A. Francalanza, and A. Ingólfsdóttir. A framework for parameterized monitorability. In *Proc. of the 21st Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'18)*, volume 10803 of *LNCS*, pages 203–220. Springer, 2018.
6. L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and S. Ö. Kjartansson. On the complexity of determinizing monitors. In *Proc. of the 22nd Int'l Conf. on Implementation and Application of Automata (CIAA'17)*, volume 10329 of *LNCS*, pages 1–13. Springer, 2017.
7. L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. Adventures in Monitorability: From Branching to Linear Time and Back Again. *PACMPL*, 3(POPL):52:1–11:29, 2019. (to appear).
8. L. Aceto, I. Cassar, A. Francalanza, and A. Ingólfsdóttir. On runtime enforcement via suppressions. In S. Schewe and L. Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
9. S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proc. of the 8th ACM European Conf. on Computer Systems (EuroSys '13)*, pages 29–42. ACM, 2013.
10. W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts: A killer application for deductive source code verification. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 1–18. Springer, 2018.
11. T. Akazaki and I. Hasuo. Time robustness in MTL and expressivity in hybrid system falsification. In *Proc. of the 27th Int'l Conf. on Computer Aided Verification (CAV'15)*, volume 9207 of *LNCS*, pages 356–374. Springer, 2015.
12. S. Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering (TSE)*, 27(8):704–714, 2001.
13. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: The SCIFF framework. *ACM Transactions on Computational Logic*, 9(4):29:1–29:43, Aug. 2008.
14. M. Althoff. Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In *Proc. the 16th Int'l Conf. on Hybrid Systems Computation and Control (HSCC'13)*, pages 173–182. ACM, 2013.
15. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
16. D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. A rule-based language for complex event processing and reasoning. In *Proc. of the 4th Int'l Conf. on Web Reasoning and Rule Systems (RR'10)*, volume 6333 of *LNCS*, pages 42–57. Springer, 2010.
17. Y. Annapureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proc. of the 17th Int'l Conf/ on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 254–257. Springer, 2011.
18. Y. S. R. Annapureddy and G. E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *Proc. of the 36th Annual Conf. on IEEE Industrial Electronics Society (IECON'10)*, pages 91–96. IEEE, 2010.

19. T. Antignac, D. Sands, and G. Schneider. Data Minimisation: A Language-Based Approach. In *Proc. of the 32nd IFIP TC Int'l Conf. on ICT Sys. Security & Privacy Protection (IFIP SEC'17)*, volume 502 of *IFIP Advances in Information and Communication Technology (AICT)*, pages 442–456. Springer, 2017.
20. C. Arackaparambil, J. Brody, and A. Chakrabarti. Functional monitoring without monotonicity. In *Proc. of the 36th Int'l Colloquium on Automata, Languages and Programming: Part I (ICALP'09)*, volume 5555 of *LNCS*, pages 95–106. Springer, 2009.
21. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
22. ARM, Cambridge, England. *ARM CoreSight Architecture Specification*, 2.0 edition, Sept. 2013.
23. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
24. S. Arshad, A. Kharraz, and W. Robertson. Identifying extension-based ad injection via fine-grained web content provenance. In *Proc. of the 19th Int'l Symp. on Research in Attacks, Intrusions, and Defenses (RAID'16)*, volume 9854 of *LNCS*, pages 415–436. Springer, 2016.
25. E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.
26. E. Asarin, O. Maler, and A. Pnueli. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 138(1):35–65, 1995.
27. E. Asarin, V. Mysore, A. Pnueli, and G. Schneider. Low dimensional hybrid systems - decidable, undecidable, don't know. *Information and Computation*, 211:138–159, 2012.
28. E. Asarin, G. Schneider, and S. Yovine. On the decidability of the reachability problem for planar differential inclusions. In *4th International Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, number 2034 in LNCS, pages 89–104. Springer-Verlag, 2001.
29. E. Asarin, G. Schneider, and S. Yovine. Algorithmic analysis of polygonal hybrid systems. part I: Reachability. *Theoretical Computer Science*, 379(1-2):231–265, 2007.
30. D. P. Attard and A. Francalanza. A monitoring tool for a branching-time logic. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 473–481. Springer, 2016.
31. D. P. Attard and A. Francalanza. Trace partitioning and local monitoring for asynchronous components. In *Proc. of the 15th Int'l Conf. on Software Engineering and Formal Methods (SEFM'17)*, volume 10469 of *LNCS*, pages 219–235. Springer, 2017.
32. H. Attiya and J. L. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. Wiley, 2004.
33. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proc. of the 6th Int'l Conf. on Principles of Security and Trust (POST'17)*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.
34. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. of the Workshop on Programming Languages and Analysis for Security (PLAS'10)*, pages 1–12. ACM, 2010.
35. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th ACM SIGPLAN-SIGACT Symp, on Principles of Programming Languages (POPL'12)*, pages 165–178. ACM, Jan. 2012.
36. S. Azzopardi, J. Ellul, and G. J. Pace. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, LNCS. Springer, 2018. To appear.
37. S. Azzopardi, G. J. Pace, F. Schapachnik, and G. Schneider. Contract automata: An operational view of contracts between interactive parties. *Artificial Intelligence and Law*, 24(3):203–243, September 2016.
38. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 1–16. ACM, 2002.
39. B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'03)*, pages 28–39. ACM, 2003.
40. H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.

41. G. Barany and J. Signoles. Hybrid Information Flow Analysis for Real-World C Code. In *Proc. of the 11th Int'l Conf. on Tests and Proofs (TAP'17)*, LNCS, pages 23–40. Springer, July 2017.
42. B. Barre, M. Klein, S.-M. Boivin, P.-A. Ollivier, and S. Hallé. MapReduce for parallel trace validation of LTL properties. In *Proc. of the 17th Int'l Conf. on Runtime Verification (RV'12)*, volume 7687 of *LNCS*, pages 184–198. Springer, 2012.
43. E. Bartocci, E. Aydin-Gol, I. Haghighi, and C. Belta. A formal methods approach to pattern recognition and synthesis in reaction diffusion networks. *IEEE Transactions on Control of Network Systems*, 5(1):308–320, 2018.
44. E. Bartocci, L. Bortolussi, M. Loreti, and L. Nenzi. Monitoring mobile and spatially distributed cyber-physical systems. In *Proc. of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'17)*, pages 146–155. ACM, 2017.
45. E. Bartocci, L. Bortolussi, L. Nenzi, and G. Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theoretical Computer Science*, 587:3–25, 2015.
46. E. Bartocci, J. V. Deshmukh, A. Donzé, G. E. Fainekos, O. Maler, D. Nickovic, and S. Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 135–175. Springer, 2018.
47. E. Bartocci and Y. Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.
48. E. Bartocci, T. Ferrére, N. Manjunath, and D. Nickovic. Localizing faults in Simulink/Stateflow models with STL. In *Proc. of the 21st ACM Int'l Conf. on Hybrid Systems Computation and Control (HSCC'18)*, pages 197–206. ACM, 2018.
49. E. Bartocci and R. Grosu. Monitoring with uncertainty. In *Proc. 3rd Int'l Workshop on Hybrid Autonomous Systems*, volume 124 of *Theoretical Computer Science*, pages 1–4. Open Publishing Association, 2013.
50. E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. In *RV 2012*, pages 168–182, 2012.
51. E. Bartocci and P. Liò. Computational modeling, formal analysis, and tools for systems biology. *PLoS Computational Biology*, 12(1), 2016.
52. D. Basin, B. Bhatt, and D. Traytel. Optimal proofs for linear temporal logic on lasso words. In *Proc. of the 16th Int'l Symp. on Automated Technology for Verification and Analysis (ATVA'18)*, volume 11138 of *LNCS*. Springer, 2018.
53. D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring of temporal specifications. *Formal Methods in System Design*, Mar. 2016.
54. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring compliance policies over incomplete and disagreeing logs. In *Proc. of the 4th Int'l Conf. on Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 151–167. Springer, 2013.
55. D. Basin, F. Klaedtke, and E. Zalinescu. The MonPoly monitoring tool. In *An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017)*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.
56. D. Basin, S. Krstić, and D. Traytel. Almost event-rate indepedent monitoring of metric dynamic logic. In *Proc. of the 17th Int'l Conf. on Runtime Verification (RV'17)*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
57. D. A. Basin, B. N. Bhatt, and D. Traytel. Almost event-rate independent monitoring of metric temporal logic. In *Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17): Part II*, volume 10206 of *LNCS*, pages 94–112. Springer, 2017.
58. D. A. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring. In *Proc. of 14th Int'l. Conf. on Runtime Verification (RV'14)*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.
59. D. A. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. MONPOLY: Monitoring usage-control policies. In *Proc. of the Second Int'l Conf. on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 360–364. Springer, 2011.
60. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring compliance policies over incomplete and disagreeing logs. In *Proc. of the Third Int'l Conf. on Runtime Verification (RV'12)*, volume 7687 of *LNCS*, pages 151–167. Springer, 2012.

61. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
62. D. A. Basin, F. Klaedtke, S. Müller, and E. Zalinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.
63. D. A. Basin, F. Klaedtke, and E. Zalinescu. Failure-aware runtime verification of distributed systems. In *Proc. of the 35th IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'15)*, volume 45 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 590–603, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
64. D. A. Basin, F. Klaedtke, and E. Zalinescu. Runtime verification of temporal properties over out-of-order data streams. In *Proc. of the 29th Int'l Conf. on Computer Aided Verification (CAV'17)*, volume 10426 of *LNCS*, pages 356–376. Springer, 2017.
65. A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proc. of the 18th Int'l Symp. on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.
66. A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1–2):49–93, 2016.
67. A. K. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering Methodology*, 20:14:1–14:64, 2011.
68. L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *Proc. of the 22nd Annual Network and Distributed System Security Symp. (NDSS'15)*. The Internet Society, 2015.
69. BBC Technology. Ransomware cyber-attack threat escalating – Europol. `https://www.bbc.com/news/technology-39913630`, 2017.
70. C. Belta, B. Yordanov, and E. A. Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.
71. A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35(3):407–427, 1999.
72. M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstic, and P. S. Pietro. Efficient large-scale trace checking using MapReduce. In *Proc. of the 38th Int'l Conf. on Software Engineering (ICSE'16)*, pages 888–898. ACM, 2016.
73. A. Bessani, M. Santos, J. ao Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proc. of the 2013 USENIX Conf. on Annual Technical Conference (ATC'13)*, pages 169–180. USENIX Association, 2013.
74. F. Besson, N. Bielova, and T. P. Jensen. Hybrid monitoring of attacker knowledge. In *Proc. of the IEEE 29th Computer Security Foundations Symposium (CSF'16)*, pages 225–238. IEEE, June 2016.
75. B. Beyer and R. Ewaschuk. *Monitoring Distributed Systems.* O'Reilly Media, Inc., Cambridge, MA, USA, 2016.
76. K. Bhargavan, N. Swamy, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, and T. Sibut-Pinote. Formal verification of smart contracts. In *Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS'16)*, pages 91–96. ACM Press, 2016.
77. D. Bianculli, C. Ghezzi, and S. Krstic. Trace checking of Metric Temporal Logic with aggregating modalities using MapReduce. In *Proc. of the 12th Int'l Conf. on Software Engineering and Formal Methods (SEFM'14)*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.
78. N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *Proc. of the 7th Int'l Conf. on Principles of Security and Trust (POST'16)*, volume 9635 of *LNCS*, pages 46–67. Springer, Apr. 2016.
79. B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, and C. Travers. Challenges in fault-tolerant distributed runtime verification. In *Proc. of the 7th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA'16): Part II*, volume 9952 of *LNCS*, pages 363–370. Springer, 2016.
80. B. Bonakdarpour, C. Sánchez, and G. Schneider. Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification. In *8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation - Track: A Broader View on Verification: From Static to Runtime and Back (ISoLA'18, part II)*, volume 11245 of *LNCS*, pages 8–27. Springer, 2018.
81. L. Bortolussi, D. Milios, and S. Guido. U-Check: Model checking and parameter synthesis under uncertainty. In *Proc. of the 12th Int'l. Conf. on Quantitative Evaluation of Systems (QEST'15)*, volume 9259 of *LNCS*, pages 89–104. Springer, 2015.

82. R. S. Boyer and J. S. Moore. *Automated Reasoning: Essays in Honor of Woody Bledsoe*, chapter MJRTY—A Fast Majority Vote Algorithm, pages 105–117. Springer, 1991.
83. P. Buiras, D. Vytiniotis, and A. Russo. Hlio: Mixing static and dynamic typing for information-flow control in haskell. In *Proc. of the 20th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'15)*, pages 289–301. ACM, 2015.
84. D. Bundala and J. Ouaknine. On the complexity of temporal-logic path checking. In *Proc. of 41st Int'l Colloquium on Automata, Languages, and Programming (ICALP'14). Part II*, volume 8573 of *LNCS*, pages 86–97. Springer, 2014.
85. A. Bünzli and S. Höfler. Controlling ambiguities in legislative language. In *Proc. of the Int'l Workshop on Controlled Natural Language (CNL'10)*, volume 7175 of *LNCS*, pages 21–42. Springer, 2010.
86. M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, pages 335–350. USENIX Association, 2006.
87. V. Buterin. A next generation smart contract & decentralized application platform. Ethereum White Paper. Available online from `https://github.com/ethereum/wiki/wiki/White-Paper`, 2017.
88. J. J. Camilleri, M. R. Haghshenas, and G. Schneider. A web-based tool for analysing normative documents in English. In *Proc. of the the 33rd ACM/SIGAPP Symposium On Applied Computing–Software Verification and Testing track (SAC-SVT'18)*. ACM, 2018.
89. J. J. Camilleri, G. J. Pace, and M. Rosner. Controlled natural language in a game for legal assistance. In *Proc. of the 2nd Int'l Workshop on Controlled Natural Language (CNL'10)*, volume 7175 of *LNCS*, pages 137–153. Springer, 2010.
90. J. J. Camilleri, G. Paganelli, and G. Schneider. A CNL for contract-oriented diagrams. In *Proc. of the 4th Int'l Workshop on Controlled Natural Language (CNL'14)*, volume 8625 of *LNCS*, pages 135–146. Springer, 2014.
91. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulleting*, 38(4):28–38, 2015.
92. I. Cassar and A. Francalanza. Runtime adaptation for actor systems. In *Proc. of the 6th Int'l Conf. on Runtime Verification (RV'15)*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.
93. I. Cassar and A. Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *Proc. of the 12th Int'l Conf. on Integrated Formal Methods iFM'16*, volume 9681 of *LNCS*, pages 176–192, Berlin, 2016. Springer.
94. I. Cassar, A. Francalanza, and S. Said. Improving runtime overheads for detecter. In *Proc. of the 12th Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA'15)*, volume 178 of *EPTCS*, pages 1–8, 2015.
95. M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computing Systems*, 20(4):398–461, Nov. 2002.
96. A. Cavoukian. Privacy by design: The 7 foundational principles. *Information and Privacy Commissioner of Ontario, Canada*, 2009.
97. A. Chakarov, S. Sankaranarayanan, and G. E. Fainekos. Combining time and frequency domain specifications for periodic signals. In *Proc. of the 2nd Int'l Conf. on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 294–309. Springer, 2011.
98. E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Proc. of the 19th Int'l Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *LNCS*, pages 474–486. Springer, 1992.
99. H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for online predicate detection. In *Proc. of the 32nd IEEE Symp. on Reliable Distributed Systems, (SRDS'13)*, pages 101–110. IEEE, 2013.
100. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proc. of the 15th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.
101. X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Proc. of the 25th Int'l Conf. on Computer Aided Verification (CAV'13)*, volume 8044 of *LNCS*, pages 258–263. Springer, 2013.
102. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, Sept. 2010.

103. C. Colombo, J. Ellul, and G. J. Pace. Contracts over smart contracts: Recovering from violations dynamically. In *Proc. of the 8th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18)*, LNCS. Springer, 2018.
104. C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1–2):109–158, 2016.
105. C. Colombo, A. Francalanza, R. Mizzi, and G. J. Pace. polyLarva: Runtime verification with configurable resource-aware monitoring boundaries. In *Proc. of the 10th Int'l Conf. on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 218–232. Springer, 2012.
106. C. Colombo and G. J. Pace. Monitor-oriented compensation programming through compensating automata. *ECEASST*, 58:1–12, 2013.
107. C. Colombo and G. J. Pace. Recovery within long running transactions. *ACM Computing Surveys*, 45, 2013.
108. C. Colombo, G. J. Pace, and P. Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
109. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*, pages 163–173. ACM, 1991.
110. G. Cormode. Continuous distributed monitoring: A short survey. In *Proc. of the First Int'l Workshop on Algorithms and Models for Distributed Event Processing (AlMoDEP'11)*, volume 585 of *ACM International Conference Proceeding Series*, pages 1–10, New York, NY, USA, 2011. ACM.
111. G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proc. of the 31st Int'l Conf. on Very Large Data Bases (VLDB'05)*, pages 13–24. VLDB Endowment, 2005.
112. G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.
113. G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *Proc. of the 2005 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'05)*, pages 25–36, New York, NY, USA, 2005. ACM.
114. G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms*, 7(2):21:1–21:20, 2011.
115. G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Continuous sampling from distributed streams. *Journal of the ACM*, 59(2):10:1–10:25, 2012.
116. G. Coulouris. *Distributed Systems: Concepts and Design.* Addison-Wesley, 2011.
117. Council of European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council. `http://eur-lex.europa.eu/legal-content/en/ALL/?uri=CELEX:32016R0679`.
118. T. Dang, A. Donzé, and O. Maler. Verification of analog and mixed-signal circuits using hybrid system techniques. In *Proc. of the 5th Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 21–36. Springer, 2004.
119. T. Dang, C. Le Guernic, and O. Maler. Computing reachable states for nonlinear biological models. In *Proc. of the 7th Int'l Conf. on Computational Methods in Systems Biology (CMSB'09)*, volume 5688 of *LNCS*, pages 126–141. Springer, 2009.
120. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proc. of the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE CS Press, 2005.
121. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of Internet packet streams with limited space. In *Proc. of the 10th Annual European Symp. on Algorithms (ESA'02)*, volume 2461 of *LNCS*, pages 348–360. Springer, 2002.
122. D. E. R. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
123. D. E. R. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
124. DEON conferences. `https://deon2018.sites.uu.nl/`.
125. J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, J. Garvit, and S. A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 1/2017, 2017.

126. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 31st IEEE Symposium on Security and Privacy (SP'10)*, pages 109–124. IEEE, 2010.
127. R. J. Dias, V. Pessanha, and J. M. Lourenço. Precise detection of atomicity violations. In *Hardware and Software: Verification and Testing*, volume 7857 of *LNCS*, pages 8–23. Springer, Nov. 2013. HVC 2012 Best Paper Award.
128. R. J. D. D. Distefano, J. C. Seco, and J. M. Lourenço. Verification of snapshot isolation in transactional memory Java programs. In J. Noble, editor, *Proc. of the 26th European Conf. on Object-Oriented Programming (ECOOP'12)*, volume 7313 of *LNCS*, pages 640–664. Springer, 2012.
129. A. Dokhanchi, B. Hoxha, and G. E. Fainekos. On-line monitoring for temporal logic robustness. In *Proc. of the 5th Int'l Conf. on Runtime Verification (RV'14)*, LNCS, pages 231–246. Springer, 2014.
130. A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. of the 22nd Int'l Conf. on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 167–170. Springer, 2010.
131. A. Donzé, T. Ferrère, and O. Maler. Efficient robust monitoring for STL. In *Proc. of the 25th Int'l Conf. on Computer Aided Verification (CAV'13)*, volume 8044 of *LNCS*, pages 264–279. Springer, 2013.
132. A. Donzé, B. Krogh, and A. Rajhans. Parameter synthesis for hybrid systems with an application to Simulink models. In *Proc. of the 12th Int'l Conf. on Hybrid Systems: Computation and Control (HSCC'09)*, volume 5469 of *LNCS*, pages 165–179. Springer, 2009.
133. A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *Proc. of the 8th Int'l Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'10)*, volume 6246 of *LNCS*, pages 92–106. Springer, 2010.
134. A. Donzé, O. Maler, E. Bartocci, D. Ničković, R. Grosu, and S. A. Smolka. On temporal logic and signal processing. In *Proc. of the 10th Int'l Symp. on Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 92–106. Springer, 2012.
135. C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Series on Integrated Circuits and Systems. Springer, 2006.
136. A. El-Hokayem and Y. Falcone. Monitoring decentralized specifications. In *Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'17)*, pages 125–135. ACM, 2017.
137. A. El-Hokayem and Y. Falcone. On the monitoring of decentralized specifications semantics, properties, analysis, and simulation. *CoRR*, abs/1808.02692, 2018.
138. J. Ellul and G. J. Pace. Runtime verification of Ethereum smart contracts. In *Proc. of the 1st Int'l Workshop on Blockchain Dependability, in conjunction with EDCC'18*, IEEE, 2018.
139. S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proc. of the 24th IEEE Symp. on Reliable Distributed Systems (SRDS'05)*, pages 73–84. IEEE Computer Society, 2005.
140. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 2003.
141. G. E. Fainekos and K. C. Giannakoglou. Inverse design of airfoils based on a novel formulation of the ant colony optimization method. *Inverse Problems in Engineering*, 11(1):21–38, 2003.
142. G. E. Fainekos, A. Girard, and G. J. Pappas. Temporal logic verification using simulation. In *Proc. of the 4th Int'l Conf. on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, volume 4202 of *LNCS*, pages 171–186. Springer, 2006.
143. G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4279–4291, 2009.
144. Y. Falcone, T. Cornebize, and J. Fernandez. Efficient and generalized decentralized monitoring of regular languages. In *Proc. of 34th IFIP Int'l Conf. on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'14)*, volume 8461 of *LNCS*, pages 66–83. Springer, 2014.
145. Y. Falcone, J. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In S. Bensalem and D. A. Peled, editors, *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.

146. Y. Falcone, J. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 14(3):349–382, 2012.
147. Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Software and System Modeling*, 14(1):173–199, 2015.
148. P. Faymonville and M. Zimmermann. Parametric linear dynamic logic. In *Proc. of the 5th Int'l Symp. on Games, Automata, Logics and Formal Verification (GandALF'14)*, volume 161 of *EPTCS*, pages 60–73, 2014.
149. S. Feng, M. Lohrey, and K. Quaas. Path checking for MTL and TPTL over data words. In I. Potapov, editor, *Proc. of the 19th Int'l Conf. on Developments in Language Theory (DLT'15)*, volume 9168 of *LNCS*, pages 326–339. Springer, 2015.
150. T. Ferrère, O. Maler, and D. Nickovic. Trace diagnostics using temporal implicants. In *Proc. of the 13th International Symposium on Automated Technology for Verification and Analysis (ATVA'15)*, volume 9364 of *LNCS*, pages 241–258. Springer, 2015.
151. T. Ferrère, O. Maler, D. Ničković, and D. Ulus. Measuring with timed patterns. In *Proc. of the 27th Int'l Conf. on Computer Aided Verification (CAV'15)*, volume 9207 of *LNCS*, pages 322–337. Springer, 2015.
152. P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. of the 2007 Int'l Conf. on Analysis of Algorithms (AOFA'07)*. DMTCS, 2007.
153. P. Fraigniaud, S. Rajsbaum, and C. Travers. On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In *Proc. of the 5th Int'l Conf. on Runtime Verification (RV'14)*, volume 8734 of *LNCS*, pages 92–107. Springer, 2014.
154. A. Francalanza. A Theory of Monitors - (extended abstract). In *Proc. of the 19th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'16)*, volume 9634 of *LNCS*, pages 145–161. Springer, 2016.
155. A. Francalanza. Consistently-Detecting Monitors. In R. Meyer and U. Nestmann, editors, *Proc. of the 28th Int'l Conf. on Concurrency Theory (CONCUR'17)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
156. A. Francalanza, L. Aceto, and A. Ingolfsdottir. Monitorability for the Hennessy–Milner Logic with Recursion. *Formal Methods in System Design*, pages 1–30, 2017.
157. A. Francalanza, A. Gauci, and G. J. Pace. Distributed system contract monitoring. *Journal of Logic and Algebraic Programming*, 82(5–7):186–215, 2013.
158. A. Francalanza and M. Hennessy. A theory for observational fault tolerance. *Journal of Logic and Algebraic Programmming*, 73(1-2):22–50, 2007.
159. A. Francalanza and M. Hennessy. A Theory of System Behaviour in the presence of Node and Link failure. *Information and Computation*, 206(6):711 – 759, 2008.
160. A. Francalanza, C. A. Mezzina, and E. Tuosto. Reversible choreographies via monitoring in erlang. In *Proc. of the 18th IFIP WG 6.1 Int'l Conf. on Distributed Applications and Interoperable Systems (DAIS'18)*, volume 10853 of *LNCS*, pages 75–92. Springer, 2018.
161. A. Francalanza, J. A. Pérez, and C. Sánchez. Runtime verification for decentralised and distributed systems. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 176–210. Springer, 2018.
162. A. Francalanza and A. Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.
163. M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.
164. G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer*, 10(3):263–279, 2008.
165. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Proc. of the 23rd Int'l Conf. on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.
166. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.
167. J. Gait. A probe effect in concurrent programs. *Software - Practise and Experience*, 16(3):225–233, Mar. 1986.

168. F. Gandon, G. Governatori, and S. Villata. Normative requirements as linked data. In *Proc. of the 30th Annual Conference on Legal Knowledge and Information Systems (JURIX'17)*, volume 302 of *Frontiers in Artificial Intelligence and Applications*, pages 1–10. IOS Press, 2017.
169. R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proc. of the 6th Conf. on Computer Systems (EuroSys'11)*, pages 107–122. ACM, 2011.
170. L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. Optimized execution of business processes on Blockchain. In *Proc. of the 15th Int'l Conf. on Business Process Management (BPM'17)*, volume 10445 of *LNCS*, pages 130–146. Springer, 2017.
171. D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: theory, implementation and applications. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proc. of the 18th ACM Conf. on Computer and Communications Security, (CCS'11)*, pages 151–162. ACM, 2011.
172. V. K. Garg. *Elements of Distributed Computing*. Wiley-IEEE Press, 2002.
173. G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proc. of the 23rd Int'l Joint Conf. on Artificial Intelligence (IJCAI'13)*, pages 854–860. IJCAI/AAAI, 2013.
174. S. Gilbertson. https://www.wired.com/2011/04/lessons-amazon-cloud-failure/, 2011.
175. J. A. Goguen and J. Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
176. O. Goldreich, editor. *Property Testing - Current Research and Surveys*, volume 6390 of *LNCS*. Springer, 2010.
177. I. Gouveia and J. Rufino. Enforcing safety and security through non-intrusive runtime verification. In *Proc. 1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems*, pages 19–24, Porto, Portugal, Dec. 2016. IEEE.
178. G. Governatori and A. Rotolo. Norm compliance in business process modeling. In *Proc. of the 4th International Web Rule Symposium: Research Based and Industry Focused (RuleML'10)*, pages 194–209, 2010.
179. R. Grigore and S. Kiefer. Selective monitoring. In *Proc. 29th Int'l Conf. on Concurrency Theory (CONCUR'18)*, volume 118 of *LIPIcs*, pages 20:1–20:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
180. S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *Proc. of the Int'l Conf. on Computer-Aided Design (ICCAD'04)*, pages 210–217. IEEE CS Press, 2004.
181. D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex event processing over streams. *CoRR*, abs/cs/0612128, 2006.
182. I. Haghighi, A. Jones, Z. Kong, E. Bartocci, R. Grosu, and C. Belta. SpaTeL: a novel spatial-temporal logic and its applications to networked systems. In *Proc. of the 18th Int'l Conf. on Hybrid Systems: Computation and Control (HSCC'15)*, pages 189–198. IEEE, 2015.
183. S. Hallé and M. Soucy-Boivin. MapReduce for parallel trace validation of LTL properties. *Journal of Cloud Computing*, 4(1):8, 2015.
184. K. Havelund and A. Goldberg. Verify your runs. In *Proc. of the First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05)*, volume 4171 of *LNCS*, pages 374–383. Springer, 2005.
185. I. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, Nov. 1989.
186. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a Javascript-like language. In *Proc. of the IEEE 28th Computer Security Foundations Symposium (CSF'15)*, pages 351–365. IEEE, 2015.
187. D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.
188. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Journal of Computer and System Sciences*, pages 373–382. ACM Press, 1995.
189. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing (PODC'03)*, pages 92–101. ACM, 2003.

190. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, May 1993.
191. S. Heule, D. Rifkin, A. Russo, and D. Stefan. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.
192. S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'06)*, pages 79–90. ACM, 2006.
193. IEEE. *IEEE P1800/D6, IEEE Approved Draft Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language*, Aug. 2012.
194. IEEE Standards Association. *IEC 62531, IEEE Std 1850 Standard for Property Specification Language (PSL)*, 2012.
195. Intel Corporation. *Control-flow Enforcement Technology Preview*, June 2016.
196. Reliable smart contracts: State-of-the-art, applications, challenges and future directions. `http://www.isp.uni-luebeck.de/Isola2018-SmartContracts`. ISoLA'18 track (`http://www.isola-conference.org/isola2018/tracks.html`).
197. W. J. B. J., D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation (NSDI'11)*, pages 141–154. USENIX Association, 2011.
198. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.
199. Journal of artificial intelligence and law. `https://link.springer.com/journal/10506`.
200. S. Jaksic, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković. From signal temporal logic to FPGA monitors. In *Proc. of the the 13th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'15)*, pages 218–227. IEEE, 2015.
201. S. Jaksic, E. Bartocci, R. Grosu, and D. Nickovic. Quantitative monitoring of STL with edit distance. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 201–218. Springer, 2016.
202. S. Jaksic, E. Bartocci, R. Grosu, and D. Nickovic. An algebraic framework for runtime verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(11):2233–2243, 2018.
203. J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, Mar. 1987.
204. J. Joyce, G. Lomow, K. Slind, and B. W. Unger. Monitoring distributed systems. *ACM Transactions on Computational Systems*, 5(2):121–150, 1987.
205. JURIX conferences. `http://jurix.nl`.
206. K. Kalajdzic, E. Bartocci, S. A. Smolka, S. D. Stoller, and R. Grosu. Runtime verification with particle filtering. In *Proc. of the 4th Int'l Conf. on Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 149–166. Springer, 2013.
207. A. Kane. *Runtime Monitoring for Safety-Critical Embedded Systems.* PhD thesis, Carnegie Mellon University, USA, Feb. 2015.
208. A. Kane, O. Chowdhury., A. Datta, and P. Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In *Proc. of the 15th Int. Conf. on Runtime Verification (RV'15)*, volume 9333 of *LNCS*, pages 102–117. Springer, 2015.
209. R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.
210. J. R. Kenny and B. Mackin. FPGA coprocessing in multi-core architectures for DSP. Altera Corporation Application Note, Sept. 2007.
211. R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proc. of the 2006 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'06)*, pages 289–300. ACM, 2006.
212. KeY. `https://www.key-project.org/applications/program-verification`, 2017.
213. E. S. Kim, S. Sadraddini, C. Belta, M. Arcak, and S. A. Seshia. Dynamic contracts for distributed temporal logic control of traffic networks. In *Prc. of the IEEE 56th Annual Conf. on Decision and Control (CDC'17)*, pages 3640–3645. IEEE, 2017.
214. S. Kong, S. Gao, W. Chen, and E. Clarke. dReach: δ-reachability analysis for hybrid systems. In *Proc. of the 21st Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, volume 9035 of *LNCS*, pages 200–205. Springer, 2015.

215. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
216. T. Kuhn. A survey and classification of controlled natural languages. *Journal of Computational Linguistics*, 40(1):121–170, 2014.
217. L. Kuhtz and B. Finkbeiner. LTL path checking is efficiently parallelizable. In *Proc. of the 36th Int'l Colloquium on Automata, Languages and Programming (ICALP'09): Part II*, volume 5556 of *LNCS*, pages 235–246. Springer, 2009.
218. L. Kuhtz and B. Finkbeiner. Efficient parallel path checking for linear-time temporal logic with past and bounds. *Logical Methods in Computer Science*, 8(4), 2012.
219. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
220. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
221. G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference - Secure Software and Related Issues (ASIAN'06)*, volume 4435 of *LNCS*. Springer, Dec. 2006.
222. J. C. Lee, A. S. Gardnerd, and R. Lysecky. Hardware observability framework for minimally intrusive online monitoring of embedded systems. In *Proc. 18th Int. Conf. on Engineering of Computer Based Systems (ECBS'11)*, pages 52–60, Las Vegas, USA, Apr. 2011. IEEE Computer Sociery.
223. J. C. Lee and R. Lysecky. System-level observation framework for non-intrusive runtime monitoring of embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 20(42):42:1–42:27, 2015.
224. L. Lessig. *Code and other laws of cyberspace*. Basic Books, 1999.
225. M. Leucker and C. Sánchez. Regular linear temporal logic. In *Proc. of The 4th Int'l Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *LNCS*, pages 291–305. Springer, 2007.
226. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic Algebraic Programming*, 78(5):293–303, 2009.
227. B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 226–238. ACM, 1991.
228. D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGSOFT Software Engineering Notes*, 2(2):128–137, Mar. 1977.
229. A. Lomuscio and M. J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.
230. D. Loreti, F. Chesani, A. Ciampolini, and P. Mello. Distributed compliance monitoring of business processes over MapReduce architectures. In *Proc. of the 8th ACM/SPEC Int'l Conf. on Performance Engineering Companion (ICPE'17)*, pages 79–84. ACM, 2017.
231. J. M. Lourenço, R. J. Dias, J. Luís, M. Rebelo, and V. Pessanha. Understanding the behavior of transactional memory applications. In *Proc. of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'09)*, pages 31–39, Chicago, Illinois, 2009. ACM.
232. D. C. Luckham. *The Power of Events - An Introduction to Complex Event Processing in Distributed Enterprise Systems*. ACM, 2005.
233. T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. of the 20th IEEE Real-Time Systems Symp. (RTSS'99)*. IEEE Computer Society, 1999.
234. Q. Luo and G. Roşu. EnforceMOP: A runtime property enforcement system for multithreaded programs. In *Proc. of the 2013 Int'l Symp. on Software Testing and Analysis (ISSTA'13)*, pages 156–166. ACM, 2013.
235. O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proc. of the Joint Int'l Confs. on Formal Modelling and Analysis of Timed Systems (FORMATS'04) and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'04)*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.
236. O. Maler and D. Nickovic. Monitoring properties of analog and mixed-signal circuits. *STTT*, 15(3):247–268, 2013.
237. A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proc of the 21st Int'l Conf. on Data Engineering (ICDE'05)*, pages 767–778, Washington, DC, USA, 2005. IEEE Computer Society.

238. A. Margara and G. Cugola. Processing flows of information: From data stream to complex event processing. In *Proc. of the 5th ACM Int'l Conf. on Distributed Event-Based Systems (DEBS'11)*, pages 359–360. ACM, 2011.
239. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
240. C. Mims. All IT jobs are cybersecurity jobs now. *Wall Street Journal*, 2017. `https://www.wsj.com/articles/all-it-jobs-are-cybersecurity-jobs-now-1495364418`.
241. J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.
242. N. Mittal, A. Sen, and V. K. Garg. Solving computation slicing using predicate detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(12):1700–1713, 2007.
243. C. Molina-Jiménez and S. K. Shrivastava. Establishing conformance between contracts and choreographies. In *Proc. of the IEEE 15th Conference on Business Informatics (CBI'13)*, pages 69–78. IEEE Computer Society, 2013.
244. R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, Oct. 1978.
245. B. Mozafari. Approximate query engines: Commercial challenges and research opportunities. In *Proc. of the 2017 ACM Int'l Conf. on Management of Data (SIGMOD'17)*, pages 521–524. ACM, 2017.
246. B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *Proc. of the 8th Biennial Conf. on Innovative Data Systems Research (CIDR'17)*, 2017.
247. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455, 2013.
248. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Bitcoin.org*, 2008.
249. T. H. Nam. *Cache Memory Aware Priority Assignment and Scheduling Simulation of Real-Time Embedded Systems*. PhD thesis, Université de Bretagne Occidentale, Brest, France, Jan. 2017.
250. H. Nazarpour, Y. Falcone, S. Bensalem, and M. Bozga. Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. *Formal Asp. Comput.*, 29(6):951–986, 2017.
251. J. Nelson. Sketching and streaming algorithms for processing massive data. *ACM Crossroads*, 19(1):14–19, 2012.
252. L. Nenzi, L. Bortolussi, V. Ciancia, M. Loreti, and M. Massink. Qualitative and quantitative monitoring of spatio-temporal properties. In *Proc. of the 6th Int'l Conf. on Runtime Verification (RV'15)*, volume 9333 of *LNCS*, pages 21–37. Springer, 2015.
253. T. Nghiem, S. Sankaranarayanan, G. E. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proc. of the 13th ACM Int'l Conf. on Hybrid Systems: Computation and Control (HSCC'10)*, pages 211–220. ACM, 2010.
254. T. Nguyen, E. Bartocci, D. Ničković, R. Grosu, S. Jaksic, and K. Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In *Proc. of 7th Int'l Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16)*, volume 9953 of *LNCS*, pages 371–379. Springer, 2016.
255. D. Nickovic, O. Lebeltel, O. Maler, and T. Ferrere. AMT 2.0: Qualitative and quantitative trace analysis with extended signal temporal logic. In *Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*, volume 10806 of *LNCS*, pages 303–319. Springer, 2018.
256. W. Orme. Debug and trace for multicore SoCs: How to build an efficient and effective debug and trace system for complex, multicore SoCs. ARM White paper, Sept. 2008.
257. L. Orosa and J. M. Lourenço. A hardware approach for detecting, exposing and tolerating high level atomicity violations. In *Proc. of the 24th Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing (PDP'16)*, pages 159–167. IEEE Computer Society, 2016.
258. G. J. Pace and G. Schneider. Challenges in the specification of full contracts. In *Proc. of the 7th Int'l Conf. on Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 292–306. Springer, 2009.
259. Y. V. Pant, H. Abbas, and R. Mangharam. Smooth operator: Control using the smooth robustness of temporal logic. In *Proc. of the 2017 IEEE Conference on Control Technology and Applications (CCTA'17)*, pages 1235–1240. IEEE, 2017.

260. R. Pardo, C. Colombo, G. Pace, and G. Schneider. An automata-based approach to evolving privacy policies for social networks. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 285–301. Springer, 2016.
261. R. Pardo, I. Kellyérová, C. Sánchez, and G. Schneider. Specification of evolving privacy policies for online social networks. In *Proc. of the 23rd International Symposium on Temporal Representation and Reasoning (TIME'16)*, pages 70–79. IEEE Computer Society, 2016.
262. R. Pardo, C. Sánchez, and G. Schneider. Timed epistemic knowledge bases for social networks. In *Proc. of the 22nd Int'l Symposium on Formal Methods (FM'18)*, volume 10951 of *LNCS*, pages 185–202. Springer, 2018.
263. R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *12th International Conference on Software Engineering and Formal Methods (SEFM'14)*, volume 8702 of *LNCS*, pages 378–392. Springer, 2014.
264. G. O. Passmore and D. Ignatovich. Formal verification of financial algorithms. In *Proc. of the 26th Int'l Conf. on Automated Deduction (CADE'17)*, volume 10395 of *LNCS*, pages 26–41. Springer, 2017.
265. P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proc. of the ACM SIGCOMM 2013 Conf. (SIGCOMM '13)*, SIGCOMM '13, pages 207–218. ACM, 2013.
266. R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *Proc. of the IEEE Real-Time Systems Symp. (RTSS'08)*, pages 481–491. IEEE Computer Society, 2008.
267. L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, pages 91–100. IEEE Computer Society, 2011.
268. P. Picazo-Sánchez, G. Schneider, and J. Tapiador. After you, please: Browser extensions order attacks and countermeasures. Under submission.
269. L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In *Proc. of the 2nd Int'l Conf. on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 310–324. Springer, 2011.
270. S. Pinisetty, D. Sands, and G. Schneider. Runtime verification of hyperproperties for deterministic programs. In *Proc. of the 6th Int'l Conf. on Formal Methods in Software Engineering (FormaliSE'18)*. ACM, 2018.
271. R. C. Pinto and J. Rufino. Towards non-invasive runtime verification of real-time systems. In *Proc. of the 26th Euromicro Conf. on Real-Time Systems - WIP Session*, pages 25–28. Euromicro, July 2014.
272. A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *Proc. of the 14th Int'l Symp. on Formal Methods (FM'06)*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.
273. H. Prakken and G. Sartor. Formalising arguments about norms. In *Proc. of the 26th Annual Conf. on Legal Knowledge and Information Systems (JURIX'13)*, volume 259 of *Frontiers in Artificial Intelligence and Applications*, pages 121–130. IOS Press, 2013.
274. C. Prisacariu and G. Schneider. A dynamic deontic logic for complex contracts. *Journal of Logic and Algebraic Programming*, 81(4):458–490, May 2012.
275. C. Prybila, S. Schulte, C. Hochreiner, and I. Weber. Runtime verification for business processes utilizing the Bitcoin Blockchain. *CoRR*, abs/1706.04404, 2017.
276. M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh. Adaptable intrusion detection using partial runtime reconfiguration. In *Proc. of the IEEE 30th Int'l Conf. on Computer Design (ICCD'12)*, pages 147–152. IEEE Computer Society, 2012.
277. V. Raman, A. Donzé, M. Maasoumy, R. M. M, A. Sangiovanni-Vincentelli, and S. A. Seshia. Model predictive control with signal temporal logic specifications. In *Proc. of the 53rd Annual Conf. on Decision and Control (CDC'14)*, pages 81–87. IEEE, 2014.
278. V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. Reactive synthesis from signal temporal logic specifications. In *Proc. HSCC'15: the 18th International Conference on Hybrid Systems: Computation and Control*, pages 239–248. ACM, 2015.
279. K. Ravichandran, A. Gavrilovska, and S. Pande. DeSTM: harnessing determinism in STMs for application development. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation (PACT)*, pages 213–224. ACM, 2014.
280. G. Reger and D. Rydeheard. From first-order temporal logic to parametric trace slicing. In *Proc. of the 6th Int'l Conf. on Runtime Verification (RV'15)*, LNCS, pages 216–232. Springer, 2015.

281. T. Reinbacher, M. Fugger, and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 24(3):203–239, June 2014.
282. T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 357–372. Springer, 2014.
283. J. Reinders. *Intel Processor Tracing.* Intel Corporation, Sept. 2013.
284. A. Rizk, G. Batt, F. Fages, and S. Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In *Proc. of the 6th Int'l Conf. on Computational Methods in Systems Biology (CMSB'08)*, volume 5307 of *LNCS*, pages 251–268. Springer, 2008.
285. A. Rodionova, E. Bartocci, D. Ničković, and R. Grosu. Temporal logic as filtering. In *Proc. of HSCC 2016: the 19th International Conference on Hybrid Systems: Computation and Control*, pages 11–20. ACM, 2016.
286. J. Rufino. Towards integration of adaptability and non-intrusive runtime verification in avionic systems. *ACM SIGBED Review*, 13(1):60–65, Jan. 2016. (Special Issue on 5th Embedded Operating Systems Workshop).
287. J. Rufino and I. Gouveia. Timeliness runtime verification and adaptation in avionic systems. In *Proc. of the 12th Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT'16)*, pages 14–20, Toulouse, France, July 2016. Euromicro.
288. RuleML conferences. `http://2018.ruleml-rr.org`.
289. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 23rd IEEE Computer Security Foundations Symposium (CSF'10)*, pages 186–199. IEEE Computer Society, 2010.
290. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
291. A. Saini, M. S. Gaur, V. Laxmi, and M. Conti. Colluding browser extension attack on user privacy and its implication for web browsers. *Computers & Security*, 63:14–28, 2016.
292. M. B. Salem, S. Hershkop, and S. J. Stolfo. A survey of insider attack detection research. In *Insider Attack and Cyber Security - Beyond the Hacker*, pages 69–90. Springer, 2008.
293. C. Sánchez and M. Leucker. Regular linear temporal logic with past. In *Proc. of the 11th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation, (VMCAI'10)*, volume 5944 of *LNCS*, pages 295–311. Springer, 2010.
294. J. F. Santos, T. Jensen, T. Rezk, and A. Schmitt. Hybrid Typing of Secure Information Flow in a JavaScript-like Language. In *Proc. of the 10th Int'l Symp. on on Trustworthy Global Computing (TGC'15)*, volume 9533 of *LNCS*. Springer, 2015.
295. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
296. N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of the 3rd ACM Int'l Conf. on Distributed Event-Based Systems (DEBS'09)*, pages 4:1–4:12. ACM, 2009.
297. K. Selyunin, S. Jaksic, T. Nguyen, C. Reidl, U. Hafner, E. Bartocci, D. Nickovic, and R. Grosu. Runtime monitoring with recovery of the SENT communication protocol. In *Proc. of the the 29th Int'l Conf. on Computer Aided Verification (CAV'17)*, volume 10426 of *LNCS*, pages 336–355. Springer, 2017.
298. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proc. of 26th Int'l Conf. on Software Engineering (ICSE 2004)*, pages 418–427. IEEE CS Press, 2004.
299. A. Shabtai, Y. Elovici, and L. Rokach. *A Survey of Data Leakage Detection and Prevention Solutions.* Springer Briefs in Computer Science. Springer, 2012.
300. J. R. E. J. Shekita and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. of the VLDB Endowment*, 4(4):243–254, Jan. 2011.
301. M. E. Shobaki and L. Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Proc. of the 2nd IEEE Int'l Symp. on Quality Electronic Design (ISQED 2001)*, pages 56–61, 2001.
302. W. Spoth, B. S. Arab, E. S. Chan, G. Dieter, A. Ghoneimy, B. Glavic, B. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, and Y. Yang. Adaptive schema databases. In *Proc. of the 8th Biennial Int'l on Innovative Data Systems Research (CIDR'17)*. CIDRDB, 2017.

303. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proc. of the 2nd Int'l Conf. on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 193–207. Springer, 2011.
304. M. Suiche. WannaCry – the largest ransom-ware infection in history. `https://blog.comae.io/wannacry-the-largest-ransom-ware-infection-in-history-f37da8e30a58`, 2017.
305. N. Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.
306. T. Todman, S. Stilkerich, and W. Luk. In-circuit temporal monitors for runtime verification of reconfigurable designs. In *Proc. of the 52nd Annual Design Automation Conference (DAC'15)*, pages 50:1–50:6. ACM, 2015.
307. P. Tsankov, S. Marinovic, M. T. Dashti, and D. A. Basin. Decentralized composite access control. In M. Abadi and S. Kremer, editors, *Proc. of the 3rd Int'l Conf. Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 245–264. Springer, 2014.
308. D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Timed pattern matching. In *Proc. of the 12th Int'l Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'14)*, volume 8711 of *LNCS*, pages 222–236. Springer, 2014.
309. D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Online timed pattern matching using derivatives. In *Proc. of TACAS'16*, volume 9636 of *LNCS*, pages 736–751, Berlin, Germany, 2016. Springer.
310. N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proc. of the 37th Int'l Symp. on Microarchitecture (MICRO'04)*, pages 243–254. IEEE Computer Society, 2004.
311. T. M. Vale, J. A. Silva, R. J. Dias, and J. M. Lourenço. Pot: Deterministic transactional execution. *ACM Transactions on Architecture and Code Optimzation*, 13(4):52:1–52:24, Dec. 2016.
312. W. M. P. van der Aalst. Distributed process discovery and conformance checking. In *Proc. of the 15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE'12)*, volume 7212 of *LNCS*, pages 1–25, Berlin, Heidelberg, 2012. Springer.
313. M. Y. Vardi. From Church and Prior to PSL. In *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *LNCS*, pages 150–171. Springer, 2008.
314. M. Viswanathan. *Foundations for the run-time analysis of software systems.* PhD thesis, University of Pennsylvania, 2000.
315. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3), Jan. 1996.
316. C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET software*, 1(5):172–179, Oct. 2007.
317. I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using Blockchain. In *Proc. of the 14th Int'l Conf. on Business Process Management (BPM'16)*, volume 9850 of *LNCS*, pages 329–347. Springer, 2016.
318. S. A. Weil, S. A.Brandt, E. L. Miller., D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, OSDI '06, pages 307–320. USENIX Association, 2006.
319. W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "next" in event processing? In *Proc. of the 26th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS'07)*, pages 263–272, New York, NY, USA, 2007. ACM.
320. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), Apr. 2008.
321. T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatation and Control*, 57(11):2817–2830, 2012.
322. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
323. D. P. Woodruff and Q. Zhang. Tight bounds for distributed functional monitoring. In *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC'12)*, pages 941–960. ACM, 2012.
324. G. H. V. Wright. Deontic logic. *Mind*, 60:1–15, 1951.
325. WSLA. `www.research.ibm.com/wsla/`.

326. A. Z. Wyner. From the language of legislation to executable logic programs. In *Logic in the Theory and Practice of Lawmaking*, volume 2 of *Legisprudence Library*, pages 409–434. Springer, 2015.
327. A. Z. Wyner, K. Angelov, G. Barzdins, D. Damljanovic, B. Davis, N. E. Fuchs, S. Höfler, K. Jones, K. Kaljurand, and T. Kuhn. On controlled natural languages: Properties and prospects. In *CNL'09*, volume 5972 of *LNCS*, pages 281–289. Springer, 2009.
328. J. Xiaoqing, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining Requirements from Closed-loop Control Models. In *Proc. of the ACM International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 43–52. ACM, 2013.
329. C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: combining ACID and BASE in a distributed database. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI'14)*, pages 495–509. USENIX Association, 2014.
330. C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *Proc. of the 25th Symp. on Operating Systems Principles (SOSP'15)*, pages 279–294. ACM, 2015.
331. S. Yaghoubi and G. Fainekos. Hybrid approximate gradient and stochastic descent for falsification of nonlinear systems. In *Proc. the 2017 American Control Conference (ACC'17)*, pages 529–534. IEEE, 2017.
332. H. Yang, B. Hoxha, and G. Fainekos. Querying parametric temporal logic properties on embedded systems. In *Proc. of the 24th IFIP WG 6.1 Int'l Conf. on Testing Software and Systems (ICTSS'12)*, volume 7641 of *LNCS*, pages 136–151. Springer, 2012.
333. J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'16)*, pages 631–647. ACM, June 2016.
334. Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys*, 50(3):41:1–41:40, June 2017.
335. K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 65(1):206–223, Jan. 2013.
336. B. Yu, Z. Duan, C. Tian, and N. Zhang. Verifying temporal properties of programs: A parallel approach. *Journal of Parallel and Distributed Computing*, 2017.
337. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 2010.