Runtime Verification for Decentralised and Distributed Systems

Adrian Francalanza[1], Jorge A. Pérez[2,3], and César Sánchez[4(✉)]

[1] CS@ICT, University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt
[2] University of Groningen, Groningen, The Netherlands
j.a.perez@rug.nl
[3] CWI, Amsterdam, The Netherlands
[4] IMDEA Software Institute, Madrid, Spain
cesar.sanchez@imdea.org

Abstract. This chapter surveys runtime verification research related to distributed systems. We report solutions that study how to monitor system with some distributed characteristic, solutions that use a distributed platform for performing a monitoring task, and foundational works that present semantics for decomposing monitors or expressing specifications amenable for distributed systems.

We will identify some characteristics that distinguish distributed monitoring from centralised monitoring, and characteristics that allow to classify distributed runtime verification works based on features of the executing platforms, the specification language and the system description. Then, we will use these characteristics to describe and compare the distributed runtime verification solutions proposed in the research literature.

Keywords: Monitoring distributed systems · Distributed monitoring
Decentralised monitoring · Monitor decomposition

1 Introduction

This chapter surveys works on runtime verification (RV) related to distributed computing systems. Distributed computing is the area of computer science devoted to the study of *distributed systems*: computational artifacts that run in execution units placed at different locations, and that exchange information using a communication infrastructure, such as a computer network (see Coulouris [38], Garg [62], Attiya and Welch [4]).

Since distributed systems encompass many different but related classes of systems, the terminology has not been uniformly used. We begin by clarifying what we mean in this chapter by different terms and conventions commonly used in distributed computing, particularly with respect to monitoring.

The computational units that form a distributed system are typically able to execute processes simultaneously, under true concurrency. Each computational

E. Bartocci and Y. Falcone (Eds.): Lectures on Runtime Verification, LNCS 10457, pp. 176–210, 2018.
https://doi.org/10.1007/978-3-319-75632-5_6

unit can run more than one *process*, and independently manage a set of local resources, typically including local memory and a local clock. We call each of these computational units a *location*.

There are two large classes of distributed systems, according to the way in which processes communicate and synchronize: systems that can use shared memory, and systems that can only use some form of message passing as means of communication. It is nowadays widely accepted to refer to the former as *parallel systems* and to the latter as distributed systems, and here we follow this convention. Additionally, some systems assume the existence of a shared clock (also called global clock) among the computational units, which is another usual classification criteria. When one assumes the existence of a global clock, the distributed system is usually called synchronous or *decentralised system*. If the global clock is not assumed then the system is called asynchronous distributed system or simply a *distributed system*. Sometimes the communication infrastructure within the distributed system is simple, as in the case of buses or broadcast communication, but it is often the case that the network *topology* is relevant for the study of a given class of distributed systems. We follow the convention that, unless specified otherwise, all execution units can talk to all other execution units directly.

In practice, components of distributed systems can fail independently. Locations are typically the units of failure, modeling crashes on the execution platform that cause all processes in the location to stop their execution. Moreover, messages in message passing systems can arrive out-of-order, be duplicated or lost, or experience unbounded delays. The nature of the failures and the high independence of failure between the different components is another factor of complexity when dealing with distributed systems. Unless stated, it is common in distributed systems to assume that the system under study presents no failures. We follow this convention here too.

Due to their concurrent nature and to the other aspects of distribution, it is well-known that distributed systems are notoriously difficult to design and reason about. Throughout the years researchers have proposed many techniques to increase the reliability of distributed algorithms and systems, including dynamic solutions. These efforts include the development of runtime verification techniques for distributed computing, which we report here. We will use *distributed runtime verification* to refer to the broad area of research that studies runtime verification in connection with distributed or decentralised systems. This includes the monitoring of distributed systems as well as the use of distributed systems for monitoring. Due again to these intrinsic difficulties, distributed runtime verification is a very active area of research and new results will be produced in the near future.

Terminology. A distributed and decentralised monitoring setting is typically built from subsystems, which we identify with processes for the discussion in this chapter. We use $P_1, P_2, \ldots$ to refer to processes. Processes execute independently and occasionally synchronize or communicate with each another via the underlying communication platform.

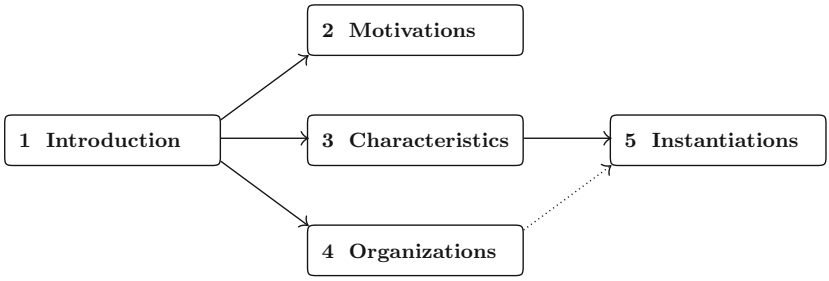
Processes are partitioned across locations, meaning that every process is located at exactly one location for any given instance. We use $l, k, \ldots$ to refer to locations. When two processes are co-located at the same location, we say that they are *local* to one another. Otherwise, we say that they are *remote*. Processes may interact and communicate with both local and remote processes. Remote communication is typically assumed to be more expensive than its local counterpart.

A *local trace* (or simply a *trace*), denoted $T_1, T_2, \ldots$ constitutes a log of past behavior used for monitoring purposes. A trace consists of a totally ordered set of trace *events*, each describing discrete computational steps of the monitored system. The ordering of trace events is necessary for the monitoring of temporal properties. A trace can describe events corresponding to a single process or else a group of processes. Although a particular location may host a number of traces (e.g., one per process hosted), we assume that a local trace cannot span across locations.

A monitoring task can be performed *online*, while the system under analysis is running, or *offline* by analysing the log after the system has finished its execution. Here we consider both kinds of solutions.

In runtime verification, monitors are created from specifications, but we will use monitoring and runtime verification interchangeably in this chapter. In online runtime verification, *monitors*—denoted as $M_1, M_2, \ldots$—are computing entities that check at runtime for the satisfaction or violation of correctness properties of the running system. Different monitors can be created to verify different properties simultaneously, and also in a modular fashion, which generally leads to better separation of concerns. The checking that the monitors perform is carried out by analysing the traces generated by the executing processes. Similar to processes, monitors are hosted by a single location for any given instance. We allow monitors to analyse multiple traces in order to generate composite traces. We do not put restrictions on whether a monitor is allowed to analyse local or remote traces, but highlight the fact that remote trace analysis may carry additional overhead costs and entail higher security risks. Monitors are allowed to communicate with one another, which gives the flexibility for property checking to be carried out in a decentralised or choreographed manner (see Sects. 4.2 and 4.4).

The rest of the chapter is organised as follows. Section 2 presents a collection of reasons that have been proposed in the literature to motivate the study of distributed runtime verification problems. Section 3 identifies a number of characteristics that are relevant in the study of the solutions proposed; these characteristics serve as a basis to classify and compare the proposed solutions. Section 4 contains a description of the different ways to organize the activities carried out by the monitoring infrastructures. Section 5 describes a collection of solutions proposed in the literature, classified according to the attributes described in the preceding sections. Finally, Sect. 6 presents current challenges and concluding remarks. The following diagram illustrates the dependencies between the sections.



Essentially, Sect. 5 contains the description and comparison of relevant work, using the classification characteristics extracted in Sect. 3.

2 Motivation and Scenarios

In this section we justify the study of distributed runtime verification. We present different scenarios that motivated research related to distributed runtime verification, according to the problem that these efforts were trying to solve. The list we present here is not intended to be exhaustive but its purpose is to give some practical justifications for the study of distributed runtime verification. Similarly, we do not claim that the papers cited are necessarily the first work to propose the study of a similar class of problems. The works mentioned below are further discussed in Sect. 5.

Observing Distributed Computations. The obvious setting where distributed monitoring arises is when the system under scrutiny is itself distributed. One important problem related to observing distributed computations is that of detecting global predicates, which is recognised as an important problem since the early ages of distributed computing (Cooper and Marzullo [37]).

It is known that checking general predicates is hard, since one has to store and enumerate all interleavings of the local processes. The so-called *computation slices* can be used for a more efficient detection (see Mittal et al. [79], Alagar and Venkatesan [1], Chauhan et al. [30]). Computation slices are abstractions of the distributed computation that guarantee the following: the predicate is present in a slice of a computation c if the predicate occurred in some state of c. This approximation is precise enough to detect the predicate. If an algorithm is too general and does not exploit the structure of the predicate under consideration, predicate detection can involve a long runtime and large memory overhead (Chauhan et al. [30]). Hence, best current solutions for predicate detection consider only fragments of the possible space of global predicates (for example the so-called linear, relational, regular and co-regular, and stable fragments) to gain efficiency. Even though most techniques for predicate detection (Cooper and Marzullo [37], Mittal et al. [79], Alagar and Venkatesan [1]) send all local events to a central process for inspection of its interleavings, some modern approaches (see Chauhan et al. [30]) consider purely distributed detection. Based on Chauhan et al. [30], Mostafa and Bonakdarpour [80] adapt the work to check whether properties defined using LTL are satisfied.

Analysis Decomposition. Most approaches to runtime verification either consider the system under dynamic evaluation as a black-box, or only inspect the internals of the monitored system with the goal to instrument the system for the monitoring task. However, it is common—using design principles like component-based design—that the description of the system is decomposed into different units.

For example, the work by Falcone et al. [49] investigates how to use the hierarchical description of the system to generate monitors that are then composed with the original system. This process produces a modified system that shares the original decomposition (and implements its functionality) and also includes the monitors embedded. Within this setting, the authors study how to compile a given design into either a centralised or a decentralised platform by deciding the placement of components using different deployment possibilities. Although the work by Falcone et al. [49] does not specifically target distributed systems, the solution obtained from the compilation of the modified system can lead to a distributed monitoring solution if the target platform is distributed.

A similar approach is exemplified by Cassar and Francalanza [23,24], where a framework for monitoring asynchronous component-based systems is presented. Again, the authors do not treat the system under scrutiny as a single monolithic block, but identify its constituent sub-components in the form of independently computing entities, called actors. The resulting monitoring setup generated is also localised to sub-components of the system, mirroring its non-monolithic structure. Even though actor systems are not necessarily executed in distributed fashion, the asynchronous nature of the code generated and its localisation lead to a straightforward distribution.

Exploiting Parallelism. Another justification for studying runtime verification in the context of distributed systems is the exploitation of parallel executing units to perform a monitoring task. For example, Berkovich et al. [14] propose to use additional hardware (a GPU parallel execution platform) to minimize the impact of online monitors on execution time overhead, reducing the intrusiveness. Moreover, the works by Francalanza and Seychell [59,60] report performance gains in terms of lower overheads when monitors are specified as concurrent entities and executed over the prevalent multi-core and multi-processor architectures. This gain is obtained because the concurrent monitors exploit better the resources of the underlying processing units.

Fault Tolerance. Handling failures in distributed systems is challenging because different components can fail independently (e.g., nodes crashing) and the communication can miss, duplicate or reorder messages or incur in unbounded delays (Francalanza and Hennessy [58]). Even worse, there can be complicated failure dependencies between components, and the resulting failure patterns can be difficult to predict and explain.

At the same time, failure tolerance can be achieved by replicating components that perform a certain task, including monitors. For example, Fraigniaud et al. [53] study the problem of distributed monitoring with failures, where events can be observed from more than one monitor, but the nodes where the monitors execute can crash. The distributed monitoring algorithm then tries to reach a verdict among the surviving monitors.

The work by Basin et al. [7] targets the incomplete knowledge caused by network failures and message corruptions and attempts to handle the resulting disagreements. A subsequent work investigates how to handle network failures, and proposes algorithms that can reach verdicts when some information is missing and messages are reordered (see Basin et al. [8]). Since message losses are also considered, this approach can also model node crashes, which are simulated by all messages from the crashed node being lost.

Efficiency. In many distributed systems scenarios, a simple monitoring solution can be obtained by implementing a central monitor that all other entities communicate with. However, distribution itself can be exploited to coordinate the monitoring task more efficiently. Many works attempt to provide more efficient solutions by exploiting the locality in the observations to also perform partially the monitoring task. For example, the works by Falcone et al. [49] and by Cassar and Francalanza [23,24], already mentioned, exploit the hierarchical structure of the system to generate local monitors. On the other hand, Cassar et al. [25] and Francalanza and Seychell [59,60] exploit the structure and semantics of the correctness property from which the monitors are synthesised to generate monitor organisations that use the underlying hardware efficiently. Concretely, the generated monitors minimize idle computing units and improve memory management via redundant monitor deallocations and monitor network reorganisations. These works pursue a more efficient monitoring where less communication and execution overhead is needed.

The pursuit of lowering overheads has also led Colombo et al. [36] to consider distribution as a means of offloading part of the monitoring computation to the computing resources of another machine. They provide handles that allow the specifier to dictate whether a property is to be runtime-checked locally, as inlined code within the monitored system, or remotely via an independent monitoring unit located on a separate machine. In separate work Colombo et al. [32] investigate various instrumentation techniques in Enterprise-Service Bus (ESB) distributed architectures, so as to determine which of them lead to lower monitoring overheads.

As observed by Bauer and Falcone [11] and in Francalanza et al. [57], when atomic observations of the monitored system occur locally, one can organize the monitors hierarchically according to the structure of the original specification. This can lead to substantial savings in communication overheads because a verdict of a subformula can often be reached further down hierarchically. From the practical point of view, Bauer and Falcone [11] claim that many cyber-physical systems, like distributed systems found in the automotive and avionics industries,

fulfill the requirement that both observations and their placement to local nodes are known at deployment time.

In the context of multithreaded programs with shared memory, the work of Luo and Roşu [73] proposes to decompose a given property into local decentralised monitors for each of the threads, which again helps to reduce monitoring overheads.

Monitoring Expressivity. Some approaches borrow directly monitoring languages from non-distributed computing, and study how to exploit or adapt the methods for distributed systems. Other approaches present new formalisms or extend existing ones with specific capabilities for distributed systems. For example, Sen et al. [89,90] propose a method to check for violations of safety properties in distributed systems, using a variation of LTL that is suitable to describe (past time) properties of distributed systems. This extension essentially allows to express the knowledge of particular agents. The work in Francalanza et al. [57] proposes and formalizes a migrating monitor setup so as to better handle the open-ended and dynamic nature of distributed systems. This helps monitoring to adapt to locations that are learnt dynamically and to varying correctness specifications over the course of long-running distributed computations.

The efficiency of migrating monitors is investigated by Bauer and Falcone [11] for fixed-location setups. The subsequent work Colombo and Falcone [34] extends these results and compares them to choreographic solutions (see Sect. 4.4).

Testing and Enforcement. Testing multithreaded programs is in general a challenging task because often concurrency errors arise only under specific interleavings and execution conditions, which are hard to cause and reproduce due to the non-determinism introduced by the scheduler. The work by Luo and Roşu [73], already mentioned, presents an *enforcement* mechanism that exploits user-specified properties to generate local monitors that can influence the executions. This approach either (1) attempts to improve testing by forcing promising schedules that can lead to violations; or (2) prevents violations of the specified property by blocking individual threads whose execution may lead to a violation. This kind of enforcement is otherwise typically implemented using ad-hoc manual synchronisation. The monitoring generation described in [73] includes the decomposition of the property into local decentralised monitors for each of the threads.

3 Characteristics of Distributed Runtime Verification

In this section we capture some challenges that distributed systems impose on monitoring and the main difficulties that must be tackled by solutions to distributed runtime verification. We begin in Sect. 3.1 by describing some key characteristics of distributed system monitoring, particularly following a historical perspective. Overall, we consider 14 characteristics, denoted (C1)–(C14). Some of

them (in particular (C1)–(C5)) are common to most distributed monitoring solutions, but are not typically a concern for non-distributed systems. Other criteria are not oblivious to all distributed monitoring cases, but identify aspects that will allow us to extract some classification dimensions, according to the approach taken by each solution. Most of these characteristics are also either unique to distributed systems or more challenging and important in distributed systems than in non-distributed systems. The classification aspects are listed later in Sect. 3.2.

3.1 Common Characteristics

Already in the late 1980s, Joyce et al. [70] identified five issues in monitoring distributed systems, in an early attempt to characterize the key constraints that distinguish monitoring in sequential settings from monitoring in distributed systems:

(C1) The fact that distributed systems have *many foci of control*;
(C2) The presence of *communication delays* among nodes, which makes it difficult to determine a system's state at any given time;
(C3) The inherent *non-determinism* in distributed and asynchronous systems;
(C4) The fact that monitoring a distributed system *alters its behavior*;
(C5) The *complexity of the interactions* between the system and the system developer.

Aspect (C1) captures the idea that a distributed system is composed of processes running independently in distributed execution units. Issue (C2) refers to one of the aspects of message passing systems. We will later refer to this aspect that allows to distinguish between systems that are not synchronised (see **Global Clock** below) and where messages can be unboundedly delayed or be lost (see **Failures** below). Not all current research in distributed monitoring assumes that messages can suffer independent delays. Issue (C3) refers to the non-deterministic and asynchronous nature common to many distributed systems. Issue (C4) refers to the intrusiveness of monitoring in the system under analysis, which is not a unique characteristic of monitoring distributed systems. We consider here intrusiveness as a key characteristic (see **Intrusiveness** below). Finally, issue (C5) refers to the additional complexity (when compared with non-distributed systems) for the engineer exercising the monitoring infrastructure, in terms of deploying the monitors and collecting and analysing the reported data. We do not develop (C5) further in this chapter as we focus on runtime verification, and not on software engineering aspects.

Another work that explores monitoring distributed systems and identifies common and classifying criteria, by Francalanza et al. [56,57], extracts the following characteristics:

(C6) Difficulties in keeping a *global state*;
(C7) Confidentiality of the information collected and communicated;

(C8) Trace analysis *locality*;
(C9) *Dynamic* aspects of specifications;
(C10) Locations constitute *units of failure*.

Maintaining a global state in a distributed system under observation is impractical for several reasons, captured by aspect (C6). One reason is that sometimes it is even theoretically impossible to build and maintain a global view, due to the lack of global clocks, asynchrony, message loss and reordering, etc. Even when it is theoretically possible, it is common that the volume of event messages that are required to build such a global view would substantially increase the monitoring overhead, making it impractical. Most works recognize that although such a central solution would greatly simplify monitoring, it is either too complex or too intrusive. This difficulty will be captured as **Global Clock** and **Failures** below.

Aspect (C7) is related to security (also mentioned by Falcone et al. [46]). Every time a trace of events is communicated across locations, the confidentiality of the information contained may be compromised. Solutions that encode and decode this information can further increase the monitoring overhead. However, we will not discuss this security aspects in this chapter.

Aspect (C8) refers to where the monitors are placed and where the events from the observed system are collected. Ideally, local monitors should analyse events locally and then communicate analysis summaries across locations. On the other hand, placement sometimes involves additional restrictions. For example, certain locations may not allow monitoring to be carried out locally due to resource constraints. Placement is often at odds with locality, which sometimes involves dynamic aspects. There are cases when it is difficult to anticipate the location where certain computations will be executed because this location depends on some runtime information that is hard to infer statically. Aspect (C8) is related to the distribution of the monitoring process, and in particular refers to the preference of decentralizing it (see **Centralisation** below).

Aspect (C9) considers that in long-running applications without a central authority, correctness specifications may not be all available prior to deployment. Some specifications are added at runtime, while the system is already executing, which disables the static placement of monitors. Dynamic aspects of monitoring are considered in (C8) and (C9), caused by either unpredictable aspects at deployment time, or constraints in the execution platform which restrict installing monitors dynamically. Finally, aspect (C10) considers again the issue of failures (see **Failures** below).

In a recent short paper, Bonakdarpour et al. [20] discuss the following four issues as distinctive, characteristic challenges of distributed runtime verification:[1]

(C11) *Modeling* a distributed RV system (particularly the system under observation);

[1] The distributed RV considered in Bonakdarpour et al. [20] is a general monitoring solution that runs on an infrastructure that is unreliable and unable to solve consensus.

(C12) Defining and evaluating distributed correctness *specifications*;
(C13) Using different *verdicts* on the state of the monitored system;
(C14) Giving semantics to the different verdicts.

Aspect (C11) concerns both the actual implementation of a distributed systems's description (including whether it is used in the monitoring process, see issue **Exploiting System Description** below), as well as efforts devoted to describing the monitoring solutions (see Sect. 4). It is well-known that describing precisely the semantics of distributed systems is more difficult than when centralised systems are considered. Aspect (C12) is related to the formalism used to describe monitors (see **Distributed Specifications**). Finally, the last two issues (C13) and (C14) are more specific to the solution provided in [20]. The first issue (C13) states that local monitors need to emit verdicts from richer domains, not just Boolean values, due to the necessary amount of information that needs to be collected and combined. This aspect has already been witnessed in monitoring non-distributed systems using LTL_3 (see Bauer et al. [13]), where the semantics of LTL for finite traces is expressed using a 3 valued domain (the third value captures the possibility of expressing an *unknown* verdict, which may become later true or false when new observations are made). Issue (C13) refers to the use of multi-valued domains as verdicts emitted the local monitors in the distributed systems. Issue (C14) refers to how these multiple verdicts can be combined during the creation of a final verdict.

3.2 Distinguishing Characteristics

We now list six dimensions that will allow us to distinguish the different lines of research and classify the solutions proposed.

Exploiting System Description. Most work in RV focuses on building monitors that can analyse any system (under some general assumptions), that is, the system is consider as a black-box that emits the necessary signals to the monitors. On the other hand, some other approaches exploit the system's description to generate specialised monitors. Examples of system's descriptions proposed include models of the system, abstractions or even full descriptions as programs. In this case, the monitors generated are only guaranteed to be correct for the specific system analysed, and in case a different system is finally deployed with the monitor, the verdicts of this monitor may not be correct. On the other hand, solutions that consider the system as a black-box generate monitors that are correct for every system (that fulfills some general assumptions) at the price of potentially less efficiency. For example, algorithms that generate monitors as finite state machines from LTL specifications work for all systems as sources of traces. If the monitor can rule out certain paths using concrete facts of the system under observation, obtained by static analysis for example, then the monitor can be specialised into a smaller finite state machine.

In some cases only certain aspects of the system description are used to build the solution, like the number of distributed nodes, the location of the individual predicates emitted by the running system, or the topology of the network.

Centralisation. Even if the system under observation is intrinsically distributed, the monitoring task can be performed in a central location that collects information from the remote units. However, solutions with a *central monitor* have many drawbacks from the points of view of overhead, efficiency, tolerance to faults and security. For these reasons, many solutions attempt to divide the monitors into local monitors and perform part of the monitoring activities locally, in a distributed fashion.

Global Clock. There are two large classes of distributed runtime verification techniques depending on whether it is assumed that all nodes have access or not to a global clock (or to perfectly-synchronised local clocks). In case a global clock is assumed, the system under analysis is equivalent to a synchronous system (following distributed computing terminology). In this case, we call the problem *decentralised monitoring*. Similarly, when monitors do not have access to a global clock we refer to the problem as *distributed monitoring*. Another characteristic feature of monitoring distributed systems is *asynchrony*, both between the monitors and the distributed system under scrutiny, and among the distributed monitors themselves.

Monitoring a distributed system often amounts to monitoring a message passing system. We reserve the term non-distributed systems for those systems that have a global clock and direct access shared memory between all computational units. For example, parallel systems (as defined above) are non-distributed systems with several concurrent execution units.

Distributed Specifications. One key classification criteria is whether the specification language from which monitors are generated has specific features for distributed systems, that is whether the formalism allows to refer to characteristics of the distributed platform. Some approaches borrow directly a language originally proposed for non-distributed systems, like LTL, and attack the problem of monitoring distributed systems against specifications written in this language. Other approaches start by introducing a modified specification language with some distributed feature, and then develop specific monitoring algorithms for this language.

Failures. In practice, both non-distributed and distributed systems are subject to failures. However, failures in distributed systems can be more subtle than in non-distributed systems due to the physical independence of the executing units. Even though most monitoring solutions assume that no component can fail, some approaches consider the possibility of some part of the distributed system failing. In particular, some of the failing aspects considered are network delays in the transmission of the messages, message loss or duplication, message corruption and node crashes. Even though Byzantine failures have been thoroughly studied in distributed systems, this aspect has received little attention in the area of monitoring distributed systems.

Intrusiveness. As already identified in early surveying efforts (see (C4) above), the monitoring process typically modifies the behavior of the monitored system. Naturally, most works focus on the effectiveness of the monitoring solution proposed, that is, on proving that the monitoring process actually detects the intended property. Some research also considers the efficiency of the combined solution (in terms of running time, number of messages, etc.) and in some few cases how the monitoring process affects the running system (that is, how intrusive monitoring is). Moreover, some works are intrusive *on purpose*, trying to reduce the intrinsic non-determinism of the running system with the goal of avoiding failures (like in enforcement) or provoking failures (for testing purposes).

4 Monitor Organisations

In this section we explain and compare the various ways in which monitoring distributed system activities can be organised. The various monitoring organisations can be explained in terms of the different configurations used to compose these components together as a monitoring infrastructure contributing towards a common goal.

The analysis of correctness properties concerning different processes, possibly spanning across different locations, often requires the aggregation of traces into composite traces. We will generally assume that the composition of two remote traces does not necessarily yield a total ordering among the events of the resulting composite trace, but instead gives a partial ordering. Monitors can communicate with each other to coordinate the monitoring task.

4.1 Traditional Monitoring

A traditional monitoring setup, depicted in Fig. 1, typically consists of a group of processes (P_1, P_2 and P_3 in the figure) that reside at one location (l). These processes generate a single local trace (T_1) that is analysed by a single monitor (M_1), also located at the same location. Even if these processes execute concurrently and are subject to a different interleaving every time the system is executed, the monitoring setup will always report a trace with a total ordering of events reflecting the executed interleaving.

4.2 Decentralised Monitoring

As depicted in Fig. 2, a decentralised monitoring setup resembles traditional monitoring in that all process executions and trace events are governed by a single global clock. Moreover, processes and monitors can communicate using synchronous channels, and computations are totally ordered. Consequently, traces can also be totally ordered, either explicitly as one data structure or locally by using time-stamps.

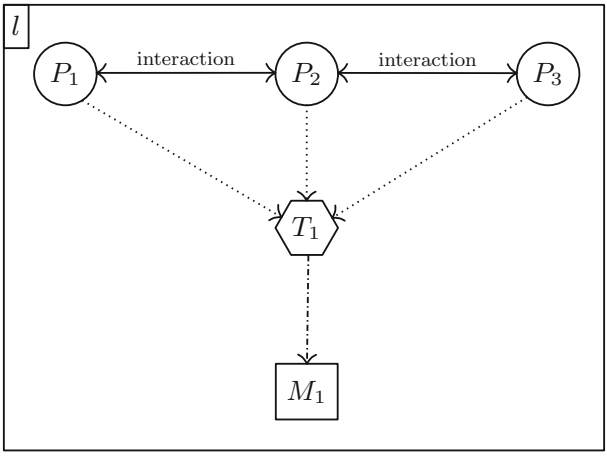


Fig. 1. A traditional (centralised) monitoring setup where processes P_1, P_2 and P_3 generate a single trace T_1 observed by a single monitor M_1. The interaction between processes illustrate that processes may communicate or synchronize, even though it is not assumed that they do (as P_1 with P_3 in the figure).

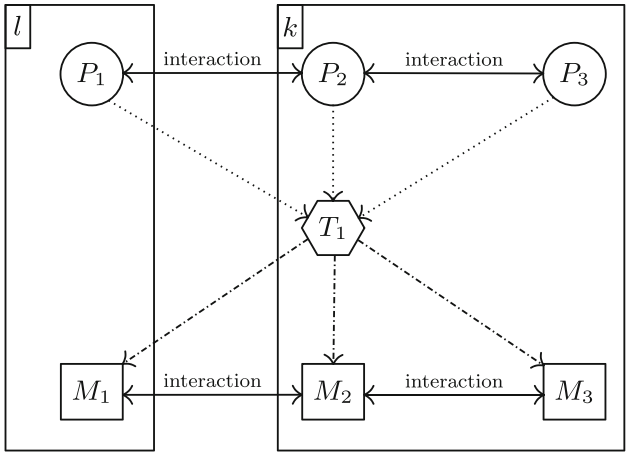


Fig. 2. In a decentralised monitoring setup the synchronised trace T_1 can be processed by several independent monitors. Now monitors can interact (like M_1 and M_2, and M_2 and M_3) but are not required to (like M_1 and M_3). Also, monitors and processes can be placed at different locations that share a global clock (l and k in the figure).

In contrast to traditional monitoring which is typically performed by a single monolithic monitor, monitoring in a decentralised and distributed setup is decomposed into different sub-components (M_1, M_2, M_3) reflecting the fact that a global correctness property may be decomposed into smaller properties.

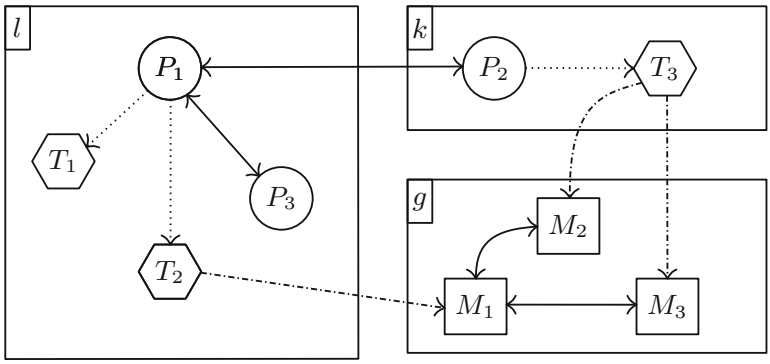


Fig. 3. In an orchestrated monitoring setup, traces are independently produced locally at the location of processes, but can be processed by remote monitors.

For instance, in cases where trace events may be attributed to different system units (e.g. classes or objects), each monitor may selectively analyse events pertaining to a particular unit entity (e.g. all the method call invocations on a particular object) and then communicate aggregate monitoring information to other monitors in order to verify a global property. It is common that sub-monitors reflect some decomposition of the specification, but sometimes sub-monitors are obtained directly by the placement of parts of the specification into locations without much decomposition.

There are also cases in which the correctness properties are inherently separate and concern only a subset of processes as in the case of parametric monitoring where the property can be evaluated independently for different parameter instances (see, e.g., Chen and Roşu [31]). In this case, monitoring may be decentralised in a natural manner without the need for the individual monitors to communicate.

4.3 Orchestrated Monitoring

Orchestrated and choreographed monitoring approaches are used in settings where more than one process is dispersed across more than one location. The set of processes generate more than one trace that can only be partially ordered due to the absence of a global clock.

In an orchestration all monitoring is ultimately performed *centrally* by a single monitor, accessing the respective trace events from different locations. The approach is depicted in Fig. 3, which shows two sub-systems located at l and k, each producing local traces of events (T_1, T_2 and T_3 respectively), subsequently analysed by monitors M_1, M_2 and M_3 from a remote location g. Each of these monitors analyse an independent correctness property.

On the one hand, the centralisation of the analysis simplifies the logic of the monitor, which is conducive to a decrease in errors in the monitor code itself. However, these benefits come at a cost in distributed settings such as the one

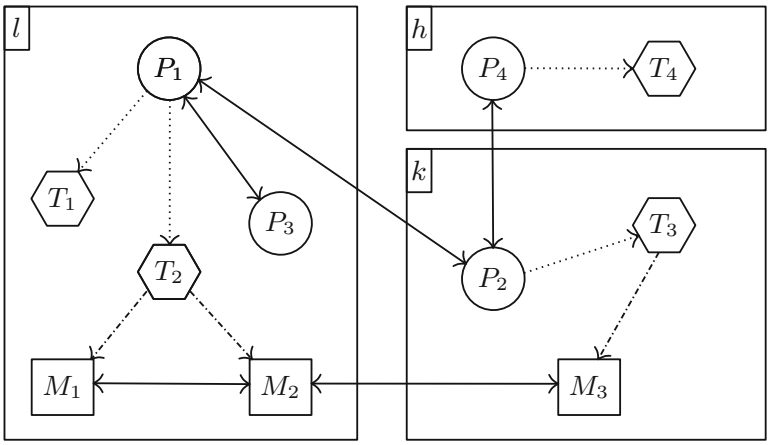


Fig. 4. In an choreographed monitoring setup, traces are independently produced locally at the location of processes and can be processed by different monitors independently.

depicted in Fig. 3. First, the approach leads to a substantial increase in the volume of trace information which has to be transmitted remotely for monitoring. The considerable increase in communication overhead across locations affects monitoring scalability when the number of processes and locations increases. The approach is also susceptible to data exposure when the trace events transmitted across locations contain private information. Adding additional security layers via mechanisms such as encryption further increases the monitoring overhead. Finally, the architecture poses a security risk by exposing the monitor as a central point of attack from which sensitive information can be tapped. Nevertheless, restricted forms of the orchestrated monitoring approach can be suitable when dealing with public information that is communicated over a relatively safe medium.

4.4 Choreographed Monitoring

A choreographed monitoring approach also targets system settings consisting of multiple processes dispersed across more than one location. In contrast to orchestrated monitoring, choreography-based approaches push the runtime verification activities locally to the location where the traces are generated, as shown in Fig. 4. The diagram depicts four processes, located at three locations l, k, and h, each generating local traces, with monitors M_1, M_2 placed at l, and M_3 placed at k. The monitor decomposition is not only due to the independence of the correctness properties being checked. In fact, monitors M_2 and M_3 could be verifying the same global property and eventually interact with each other in order to synchronize their monitoring effort.

The appeal of localizing monitoring is the potential minimisation of data exposure and communication overhead. By verifying locally, we avoid having to

transmit trace information to a remote monitor. Moreover, localised monitors typically require less communication than remote monitoring using a central monitor. However, choreography is more complex to instrument, since correctness properties need to be decomposed into coordinated local monitors. Furthermore, choreographed monitoring is also more intrusive, by burdening the monitored subsystems with additional local computation, and is thus applicable only when the hosting locations allow local instrumentation of monitoring code.

5 Instantiations

In this section we describe and compare research solutions proposed in the literature, using the characteristics captured in Sect. 3 and the organisations described in Sect. 4. To ease the description we group the papers as follows:

- Decentralised Monitoring
- Distributed Monitoring
- Fault Tolerance
- Monitor Decomposition
- Predicate Detection for Distributed Systems
- Intrusiveness
- Behavioral Type Systems for Distributed Monitoring

Other features cross-cut papers across different characteristics and are mentioned in each particular case. Tables 1 and 2 summarize the papers according to the main characteristics considered. In the characteristics shown in the tables, **Global Clock**, **Failures**, **System Description**, **Distributed Specs** and **Intrusiveness** are directly characteristics captured in Sect. 3. **Asynchronous Msgs** refers to whether the underlying platform is a message passing system. **Asynchronous Msgs**, **Decentralised Monitoring** and **Distributed Monitoring** are characteristics considered within **Global Clock** in Sect. 3.

The entries **LTL** and **Predicate Detection** are included because these specification languages have been thoroughly considered in many works. Finally, **Types** refer to a line of research based on process algebras and session types.

Decentralised Monitoring. Bauer and Falcone [11,12] study the problem of decentralised monitoring. The starting point is a specification expressed in LTL without any specific extension for distributed systems, except for the static mapping of atomic predicates to individual processes. Note that individual state predicates of the specification may be split into more than one process. The solution synthesizes a monitor for each process, under the assumption that components communicate synchronously with a global clock. Each component has a local monitor attached, and emits events synchronously after every global clock tick. By design, the solution to a verdict is taken as combination of the execution of the local monitors, lacking a central decision-making point. This work is later generalised by Falcone et al. [46] beyond LTL to cover all regular languages.

Table 1. State-of-the-art on distributed monitoring. Each paper is classified according to the characteristics considered (part 1).

Publication	year	Global Clock	Asynchonous Msgs	Decentralised Monitoring	Distributed Monitoring	Failures	System Description	Distributed Specs.	LTL	Predicate Detection	Intrusiveness	Types
Bauer and Falcone [11]	2012	✓	·	✓	·	·	·	·	✓	·	·	·
Bauer and Falcone [12]	2016	✓	·	✓	·	·	·	·	✓	·	·	·
Colombo and Falcone [33]	2014	✓	·	✓	·	·	·	·	✓	·	·	·
Colombo and Falcone [34]	2016	✓	✓	✓	·	·	·	·	✓	·	·	·
Falcone et al. [46]	2014	✓	·	✓	·	·	·	·	·	·	·	·
Bartocci [6]	2013	✓	✓	✓	·	·	·	·	✓	·	·	·
Sen et al. [89]	2004	·	✓	·	✓	·	·	✓	✓	·	·	·
Francalanza et al. [56]	2011	·	✓	·	✓	·	✓	·	·	·	·	✓
Francalanza et al. [57]	2013	·	✓	·	✓	·	✓	·	·	·	·	✓
Basin et al. [7]	2013	✓	·	·	·	✓	·	·	·	·	·	·
Basin et al. [8]	2015	✓	✓	·	✓	✓	·	·	·	·	·	·
Fraigniaud et al. [53]	2014	·	✓	·	·	✓	·	·	✓	·	·	·
Bonakdarpour et al. [19]	2016	·	✓	·	·	✓	·	·	✓	·	·	·
Falcone et al. [49]	2015	·	·	·	·	·	✓	·	·	·	·	·
Bonakdarpour et al. [17]	2010	·	✓	·	·	·	✓	·	·	·	·	·
Bonakdarpour et al. [18]	2010	·	✓	·	·	·	✓	·	·	·	·	·
Berkovich et al. [14]	2015	✓	·	·	·	·	·	·	✓	·	✓	·
Francalanza and Seychell [59]	2013	·	✓	·	✓	·	·	·	·	·	·	·
Francalanza and Seychell [60]	2015	·	✓	·	✓	·	·	·	·	·	·	·
Attard and Francalanza [3]	2016	·	✓	·	✓	·	·	·	·	·	·	·
Chase and Garg [29]	1998	·	✓	·	·	·	·	·	·	✓	·	·
Cooper and Marzullo [37]	1991	·	✓	·	·	·	·	·	·	✓	·	·
Garg and Waldecker [64]	1994	·	✓	·	·	·	·	·	·	✓	·	·
Garg and Mittal [63]	2001	·	✓	·	·	·	·	·	·	✓	·	·
Mittal and Garg [78]	2005	·	✓	·	·	·	·	·	·	✓	·	·
Mittal et al. [79]	2007	·	✓	·	·	·	·	·	·	✓	·	·
Sen and Garg [88]	2007	·	✓	·	·	·	·	·	·	✓	·	·
Ogale and Garg [83]	2007	·	✓	·	·	·	·	·	✓	✓	·	·
Chauhan et al. [30]	2013	·	✓	·	✓	·	·	·	·	✓	·	·
Mostafa and Bonakdarpour [80]	2015	·	✓	·	✓	·	·	·	✓	✓	·	·
Sen and Garg [86]	2003	·	✓	·	·	·	·	·	✓	✓	·	·
Luo and Roşu [73]	2013	✓	·	✓	·	·	✓	·	·	·	✓	·
Cassar and Francalanza [22]	2014	·	✓	·	✓	·	·	·	·	·	✓	·
Zhang et al. [93]	2016	·	✓	·	✓	·	·	·	·	·	✓	·
Colombo et al. [36]	2012	·	✓	·	·	·	·	·	·	·	✓	·

Table 2. State-of-the-art on distributed monitoring. Each paper is classified according to the characteristics considered (part 2).

Publication	year	Global Clock	Asynchonous Msgs	Decentralised Monitoring	Distributed Monitoring	Failures	System Description	Distributed Specs.	LTL	Predicate Detection	Intrusiveness	Types
Colombo et al. [35]	2011	·	✓	·	✓	·	·	·	·	·	✓	·
Cassar and Francalanza [24]	2016	·	✓	·	✓	·	·	·	·	·	✓	·
Bocchi et al. [15]	2013	·	✓	·	✓	·	·	✓	·	·	·	✓
Bocchi et al. [16]	2017	·	✓	·	✓	·	·	✓	·	·	·	✓
Hu et al. [67]	2013	·	✓	·	✓	·	·	✓	·	·	·	✓
Demangeon et al. [41]	2015	·	✓	·	✓	·	·	✓	·	·	·	✓
Neykova et al. [82]	2013	·	✓	·	✓	·	·	✓	·	·	·	✓
Neykova et al. [81]	2014	·	✓	·	✓	·	·	✓	·	·	·	✓
Jia et al. [69]	2016	·	✓	·	✓	·	·	✓	·	·	✓	✓
Di Giusto and Pérez [42]	2015	·	✓	·	✓	✓	·	✓	·	·	✓	✓
Di Giusto and Pérez [43]	2016	·	✓	·	✓	✓	·	✓	·	·	✓	✓
Castellani et al. [26]	2014	·	✓	·	✓	✓	·	✓	·	·	✓	✓
Castellani et al. [27]	2016	·	✓	·	✓	✓	·	✓	·	·	✓	✓
Mezzina and Pérez [75]	2016	·	✓	·	✓	·	·	✓	·	·	✓	✓
Mezzina and Pérez [76]	2017	·	✓	·	✓	·	·	✓	·	·	✓	✓

The main advantage of a decentralised solution over a non-distributed one is that not all events must be sent to the location of the central monitor. The challenge is that local monitoring must be performed with only partial observations of the global trace. The algorithm progresses by rewriting the specification at each node, with the partial information available. When local monitors are unable to evaluate a specification given their local view of the computation, they communicate their residual formulas to the other monitors. An alternative approach would use a central monitor that receives information about the local states of all other locations. One of the main practical concerns is how the decentralised approach compares with this alternative central approach. The empirical evaluation reported by Bauer and Falcone [11,12], Falcone et al. [46] suggests that the overhead introduced is lower in the distributed solution. There is also an economic advantage in the decentralised solution, because in a distributed solution there is no need to add a central processor. Practical applications of this approach involve monitoring the behavior of embedded systems that are distributed by nature, like cars and airplanes where the different distributed components are known upfront. These include typical field-busses like EtherCAT, ProfiBus and

ProfiNet (also known as "Industrial Ethernet" [50]). In these systems, processes communicate over a synchronous bus, so the global clock assumption is justified.

Bartocci [6] extends the work by Bauer and Falcone [11] to real-time embedded systems by considering the maximum duration of the computation and communication. The main result is the ability to calculate a sampling ratio above which the decentralised monitoring process is guaranteed to generate the correct outcome.

The works by Colombo and Falcone [33,34] start from similar assumptions and goals: there is a global clock and one local monitor per executing component. The number of executing components is also known upfront. The work in [34] removes the assumption of instantaneous communication from Colombo and Falcone [33] and enables a solution with reliable messages with any delay. Still, a global clock is assumed because the specification logic is LTL and individual predicates sensed are totally ordered. The solution proposed is a choreographed decentralised monitoring algorithm, where each local monitor senses a collection of local predicates. The local monitors use the rewriting approach (also known as formula progression) by which the state of the monitor is the LTL formula that results by expanding the LTL formula to the residual formula in the next state, simplified with the acquired knowledge. A key element in the solution is that a network of monitors is statically built by assigning each subformula of the original formula to a node in the distributed system. The hierarchical description inherent by the sub-formula relation in turn dictates the communication pattern between the local monitors. Consider a formula ψ and let φ be a sub-formula of ψ. The monitor M_1 for φ informs the monitor M_2 for ψ about the verdict of φ which, in turn, is used by M_2 to compute the verdict of ψ. If a synchronous clock is assumed, the root formula verdict is guaranteed to be reached within at most k steps of delay, where k is the height of the original formula.

Distributed Monitoring. Sen et al. [89] propose a method to detecting violations of safety properties in an asynchronous distributed system, where no global clock is assumed. The method proposed generates, given a specification, local monitors for all distributed nodes. These local monitors communicate only by piggybacking additional information in the messages sent by existing processes in the system, so the shape of the history of messages exchanged is not modified by the actions taken by the monitors. The logic used in Sen et al. [89] extends past time LTL with features for distributed systems, in particular an operator $@_j\varphi$, which captures the most recent value of formula φ according to process j.

The algorithm uses vector clocks (see Lamport [71], Mattern [74] and Fidge [51,52]) to transmit the most recent value of sub-formulas needed to compute the outcome of their containing formulas. Then, at deployment time, the monitor specification is decomposed into local monitors that collect information locally and compute the current value of formulas, based on this local information and on the information received in messages about the causal past of remote processes. This approach allows to generate monitors without inspecting the internal behavior of each process.

Francalanza et al. [56,57] present a formal model for distributed monitoring. System computations are described as π-calculus processes (Milner et al. [77]) hosted at different locations and interacting with one another via message passing. When systems compute, they generate residual trace events that are only locally ordered (with respect to the other events generated at the same location) but globally unordered (with respect to events generated at other locations), thereby modeling the absence of global clocks. Distributed monitors, also residing a different locations, are then tasked with analysing local traces and interacting with one another in order to perform a global analysis of system computation. The model is equipped with a bisimulation-based equivalence relation that is used to reason about different distributed monitoring strategies such as those discussed in Sect. 4. The model is also used to define and evaluate a new migrating monitor strategy that better handles the dynamic nature of open distributed systems.

Fault Tolerance. Not many works attack the problem of monitoring distributed systems considering that components can fail. Notable exceptions are the works by Basin et al. [7,8], and by Fraigniaud et al. [53] and Bonakdarpour et al. [19].

Basin et al. [7] present a policy language, a variant of FOLTL with three-valued semantics, and an algorithm that allows to reason about incomplete knowledge and handle disagreements. The main practical motivation is to handle errors in the observed trace, for example due to corruption or loss of part of logs files in complex IT systems, crashes in running systems, or network failures. Another motivation is to reconcile different views or verdicts obtained from monitors that observe different parts of the logs. The key idea is to equip the execution with features to enable monitors to distinguish between an event not being observed and the event not existing at all. The authors claim that any approach that solves this problem must satisfy that, once a definite verdict is given, providing more compatible information cannot retract the definite verdict. They manage to provide a complete algorithm for a fragment of the policy language. A similar work on compliance checking is Garg et al. [61]. Even though these works do not explicitly handle distributed systems, they handle runtime verification under incomplete information and incorrect information, which can be used to encode problems for distributed systems.

Influenced by [7], Basin et al. [8] consider the problem of monitoring distributed systems in the presence of network failures. The authors also consider the case that the monitor itself is distributed for the purposes of efficiency, performing the monitoring computation closer to the observation point and reducing the communication overhead. The paper [8] deals with Metric Temporal Logic (MTL), a logic that allows to express real-time properties. The algorithm is designed based on the *timed asynchronous* model for distributed systems (see Cristian and Fetzer [39]), which assumes the availability of highly-synchronised local clocks but permits crash failures in the processes and in the network. Another assumption is that components are known at deployment time.

In [8] processes time-stamp their observation before communicating them to the local monitors. The time-stamp allows components to compute precise delays between events, and to totally order the events. It is interesting to point out that even without failures, reliable asynchronous networks allow messages to arrive in different orders. Forcing messages to arrive in order requires buffering messages to ensure proper delivery order, which in turn prevents the early detection of some violations that would be possible with out-of-order delivery. The algorithm in Basin et al. [8] uses a richer value to encode the absence of knowledge when evaluating part of the specification. When the missing information is finally received, the monitor can precisely resolve the uncertainty. Sometimes, a monitor can reach a precise verdict only with the partial information received in a timely manner. Consequently, the algorithm can monitor MTL properties tolerating the out-of-order arrival of partial observations.

Concerning organisation, the monitors in [8] are distributed in a directed acyclic graph (DAG) where each monitor handles a subformula of the given formula, and children nodes handle subformulas of the formula handled by their parent node. The root of the DAG handles the original formula. During execution, messages are sent from children to parent monitors to inform about the verdicts reached in the subformula handled at the given point in time. When a process performs an atomic observation it also equips the time-stamp with an additional sequence number, which is locally unique. This sequence number allows monitors to infer the existence or absence of unknown intermediate samples between two observations. Intermediate nodes can also send heart-beat messages, which serve the purpose of informing about the absence of verdicts and the health of the intermediate node. Heart-beats also allow to infer the existence or absence of intermediate meaningful observations or verdicts, and in turn compute timeouts.

The problem of distributed monitoring for asynchronous distributed systems with node crashes is considered by Fraigniaud et al. [53] and Bonakdarpour et al. [19]. Monitors can either work correctly or fail, but after a fail, monitors do not perform any action for the reminder of the execution. The solution is based on the asynchronous "wait-free" communicating infrastructure. It is well known from the research area of distributed algorithms that the wait-free model of computation (see Attiya and Welch [4]) can simulate many other models of crash-fail asynchronous distributed systems. The main result in [19,53] is an algorithm and a lower-bound on the number of different verdicts that monitors need to communicate with each other to correctly detect the violation of an LTL property. The lower bound on the number of verdicts reveals that monitors need to communicate complex information in order to compute a global outcome. The final verdict reached by the cooperating monitors, in turn, will be that of LTL_3. The following three options are possible: (1) the property is satisfied in all continuations; (2) the property is violated in all continuations; (3) the outcome is unknown. These papers do not assume that the observations of the distributed monitors are disjoint. Even though monitors may only be observing part of the global input alphabet, several monitors may overlap in their partial observation.

Monitor Decomposition. Falcone et al. [49] target the problem of monitoring component based-systems, that is, systems that are described by the composition of components. More precisely, in [49] systems are described using the Behavior-Interaction-Priority (BIP) component-based framework (see Basu et al. [9]). Even though this paper does not attack explicitly the problem of monitoring a distributed system, it is nowadays well understood that component-based descriptions can be compiled into distributed implementations (see Bonakdarpour et al. [17,18]). Consequently, the monitors generated at the component level following [49] are attached to the system generating a modified BIP description that can subsequently be compiled into a distributed system.

Monitor decomposition for decentralised monitoring can also be inferred from the specification formula from which a monitor is synthesised. This line of research is explored extensively by Francalanza and Seychell [59,60] and Attard and Francalanza [3] for both safety and co-safety properties of logics involving conjunctions, disjunctions and recursion. Conjunctions and disjunctions are synthesised into concurrent monitors that analyse sub-parts of the system, whereas recursion leads to the dynamic generation of concurrent monitors, generated lazily only when required to minimize monitoring overheads. In every case, the concurrent monitors generated lead to self-contained localised monitoring that can be readily distributed. The automated synthesis function is proved correct in each of these cases (see Francalanza et al. [55] for the correctness proof in [3]). The work by Cassar et al. [25] considers a refined implementation where the concurrent sub-monitors cooperate among themselves and reorganize their interconnection so as to optimize the resources used for monitoring, thus reducing monitoring overheads.

Predicate Detection for Distributed Monitoring. Predicate detection (see Chase and Garg [29]) consists on checking whether a certain predicate occurred during the distributed execution, or more formally, whether the predicate holds in some *consistent cut* of the execution. In this context, predicates are state formulas (and consequently safety properties) even though some work has extended predicate detection to richer temporal formulas (see below for details).

All algorithms for predicate detection assume that the collection of executing processes is known a-priori, that processes do not fail and that all messages eventually arrive. Predicate detection can be performed offline, when all events are available before the detection algorithm starts running, or online, when one event at a time is processed. There are three main techniques for predicate detection. The first technique uses the global snapshots proposed by Chandy and Lamport [28], which can only detect *stable* predicates, which are predicates that remain true after becoming true (like termination, but unlike mutual exclusion). The second technique consists in an explicit construction of the lattice of global states proposed by Cooper and Marzullo [37]. This technique can detect unstable predicates but it is exponential in the number of local states and processes. Finally, the third technique exploits the specific structure of the predicate to provide efficient solutions. Examples include conjunctions of local predicates (Garg and

Waldecker [64]) and relational predicates of the form $\sum_i x_i < C$, where x_i are local variables.

Even if one had access to all the local histories of the execution of all processes, detecting a predicate is hard because—for general Boolean formulas—one needs to enumerate and search all possible interleavings of the local executions. Chase and Garg [29] show that detection of 2-CNF predicates is an NP-hard problem, even when assuming a central monitor. A solution to this explosion problem is a technique called *slicing* (see [63]). Slices are abstractions of the computation that guarantee that the predicate is detected in a slice if and only if the predicate holds in some consistent cut of the original computation. Computing a slice for a general predicate is still an NP-hard problem, shown by Mittal and Garg [78], but when efficient slices exist, these are much smaller than actual explicit histories. Consequently, a line of work has focused on identifying classes of predicates for efficient slicing procedures exist. These slices are based on fragments of the logic used to express the global state predicates. These fragments include regular, co-regular, linear, relational and stable predicates (see Mittal and Garg [78], Mittal et al. [79], Sen and Garg [88], Ogale and Garg [83]). Some of these solutions construct the slices offline, assuming that the whole histories are available to the slicing algorithms, while others work online, building the slice incrementally. Similarly, most of the solutions are still centralised (Cooper and Marzullo [37], Mittal et al. [79], Sen and Garg [88], Mittal et al. [79]) in the sense that all histories are sent to a central monitor that computes the slice and detects the predicate.

The first distributed solution to slice-based predicate detection is by Chauhan et al. [30]. The solution is online and distributed, in the sense that the slicing is computed by the distributed monitors. The guarantee is that if the predicate exists in a consistent cut of the computation, then it is detected by some monitor. The algorithm exploits both the structure of the property ([30] study regular properties) and epistemic information about what the knowledge that the different monitors acquire.

Also, even though most approaches are restricted to state predicates (or more precisely, fragments of the propositional logic for state predicates), some approaches tract richer temporal properties. For example, Sen and Garg [87], Ogale and Garg [83] present methods for sliced based predicate detection for a fragment of temporal logic that includes invariants (AG) and possible reachable (EF) operators, which extends the applicability beyond safety properties into a subclass of CTL formulas called Regular CTL (see Sen and Garg [86]). The restrictive use of negation in [87] is relaxed in [83]. Even though the work in [83,87] is applicable to a richer fragment of temporal logic, these algorithms work with a central monitor.

More recently, Mostafa and Bonakdarpour [80] provide a solution for monitorable LTL_3 temporal properties, but in this case extending the work of Chauhan et al. [30] so the solution obtained is distributed. This solution inserts additional messages in the network and is not restricted to only piggybacking information in existing messages.

Intrusiveness. It is often desirable that the monitoring process perturbs the execution of the system under analysis in the least possible manner. Typically, either the system is instrumented by embedding monitors in the code itself, or monitors and processes share resources because they execute in the same platform. These changes affect the behavior of the system, sometimes in a significant manner.

Berkovich et al. [14] propose to use additional hardware, and in particular a GPU parallel execution platform, to minimize the impact of online monitoring. The authors show how to generate parallel monitors from temporal logic specifications and evaluate empirically that the obtained parallel monitors together with the additional GPU hardware alleviate the effect of monitoring on the execution of the original system. This is a parallel solution (and not a message passing distributed solution) to reduce the intrusiveness of monitoring.

Other times, it is desirable that the monitoring process perturbs the execution of the system. One example is runtime enforcement, where the objective of the "monitoring" is to guarantee that the system stays within a safe region of states. Consequently, the enforcement system uses the information provided by the monitor to prevent an error before it occurs (see the chapter in this monograph about runtime enforcement). Another example is testing of multithreaded programs, which is in general a very hard task, due to the non-deterministic nature of the execution of concurrent programs, and the difficulty to reproduce erroneous behaviors. In this context it is desirable to guide the system towards executions that are more likely to produce an error. The work by Luo and Roşu [73] consists of an *enforcement* mechanism that uses user-specified monitors to generate local monitors. Such local monitors block individual threads that violate the specified properties. This enforcement pursues two objectives: (1) to guarantee the enforcement of properties in a multi-threaded program in a systematic way, which is typically implemented using ad-hoc synchronisation manually; and (2) to force schedules that test properties during the testing of multithreaded programs. The monitor generation described in Luo and Roşu [73] includes the decomposition of the property into local decentralised monitors for each of the threads.

The body of work by Roşu and Havelund [85], Cassar and Francalanza [22], Zhang et al. [93] explores the idea of decoupling the execution of monitors from the systems under scrutiny. This approach uses a mixture of synchronous and asynchronous monitoring, in order to obtain a feasible instrumentation setup that distribute monitors and systems at different locations, such as in the case of Colombo et al. [36] and other orchestrated monitoring setups. Asynchronous monitoring, used in various monitoring tools such as Colombo et al. [35], Francalanza and Seychell [60], Zhang et al. [93], Attard and Francalanza [3], minimizes monitor intrusiveness because it requires less instrumentation effort. Moreover, Cassar and Francalanza [22], Zhang et al. [93] show that this method of instrumentation can substantially reduce monitoring overhead. By using hybrid solutions, they also show how one need not compromise on the timeliness of detections.

Cassar and Francalanza [23,24] extend the concept of non-intrusiveness to runtime adaptation via hybrid asynchronous monitoring. The goal is to design monitors that intervene with the execution of the system under scrutiny, and apply these interventions (i.e. system adaptations) with minimal overheads. In particular, the work [24] implements a framework where the monitors for system components can act at varying degrees of synchrony with respect to the observed components. Some parts of the system can be executed in a decoupled fashion with their monitors when no adaptations on that sub-system are required. Later, these sub-systems can be incrementally synchronised with the respective monitor when an adaptation is about to be applied. The entire framework is implemented atop a completely asynchronous actor computational model, which eases the distribution over remote locations.

Behavioral Type Systems for Distributed Monitoring. In this subsection we describe the work in process calculus related to studying the monitoring of distributed systems. Many large-scale systems consist of heterogeneous, distributed software artifacts (processes) that interact following some precise protocols. In these communication-centric settings, processes communicate asynchronously, without a global clock, and are prone to local failures. These characteristics make distributed monitoring a suitable approach to enforce system correctness by complementing the static verification techniques that are typically applied individually to each process. As we detail next, monitoring for communication-centric systems is an instance of the choreographed monitoring organisation described in Sect. 4.

A productive research strand to the analysis of communication-centric software systems uses *process calculi* (such as the π-calculus) as minimal specification languages. These formal calculi provide an unambiguous setting in which the communication correctness of these systems can be compositionally established. In particular, coupling process calculi with so-called *behavioral type systems* allows to (statically) enforce safety and liveness properties associated to protocol conformance. Rather than classifying data values, behavioral types define abstractions of the protocols that a communication entity (say, a socket or a channel) should respect throughout its execution (see Hüttel et al. [68] for a survey).

Several works have explored the interplay of behavioral types and mechanisms for distributed monitoring. In particular, monitoring frameworks based on *session types*, a particular class of behavioral types, have been put forward. Session types organize a series of communication actions corresponding to the same reciprocal protocol into a structure called *session* (see Honda et al. [65]). While typed process frameworks for binary session types can analyse two-party protocols, more general type theories for multiparty session types cover the case of protocols with three or more participants (Honda et al. [66]). Both binary and multiparty session types start to make their way into mainstream programming languages and frameworks (Ancona et al. [2]). In the multiparty case, a global type entirely describes the intended communication scenario. By projecting this

global type onto each protocol participant, one may obtain its corresponding local type, which abstracts a participant's contribution to the protocol. This collection of local types thus offers a key reference for obtaining correct implementations for all participants.

Communication-centric systems often comprise components made available as grey- and black-boxes, with limited communication interfaces. As such, static verification techniques are unsuitable for their validation. Motivated by this observation, several works develop abstract frameworks based on process calculi in which monitors are terms of the specification language. The formal semantics of these calculi uses these monitor terms to enable process behavior according to the intended protocol. Rather than a logical specification (say, an LTL formula), each monitor uses a behavioral type (e.g., a local protocol) to guide a participant's behavior. These works define a special case of choreographed monitoring: the coupling of processes and monitors at the same level of abstraction makes the notion of local trace implicit. Monitors do not communicate to each other, nor perform autonomous actions. The global type through its projections is used to synthesize a monitor for each participant. This way, even untyped processes can be used to implement a protocol participant as long as they offer the right communication actions at the right time, in accordance with the governing local protocols.

Based on this general setup, Bocchi et al. [15,16] develop a monitored π-calculus with dynamic usage of multiparty session types, offering local and global safety assurance of distributed components. In their model, a network is a collection of processes (one per participant) that communicate via asynchronous message passing. Each participant is equipped with a trusted monitor that guards the run-time behavior of both the principal and its environment—this is realised by the evaluation of incoming and outgoing messages. Monitors regulate the creation of sessions and movement of messages within sessions. This dynamic checking can be switched off when processes have been statically verified. A series of queues shared between principals is assumed to support message passing, together with a global transport that abstracts distributed communication.

Building upon Bocchi et al. [15] and Bocchi et al. [16], the works by Hu et al. [67], Demangeon et al. [41] propose a dynamic verification framework for multiparty session types that admit interruptions. This a practical framework, which relies on the Scribble protocol language, an implementation of multiparty session types (see Yoshida et al. [92]), to specify global protocols, and on a Python API for conversation programming. In this framework, the monitor that tracks the progress of each participant within a session is represented using a finite state machine (FSM), generated from the local type. By independently monitoring each session endpoint at runtime, this framework ensures global communication safety even in the presence of asynchronous interruptions.

Other works on a practical strand are [81,82]. Neykova et al. [82] propose a toolchain for designing deadlock-free multiparty global protocols. Using automatically generated monitors for each session endpoint, this toolchain can detect illegal communication actions and mistaken message types that go against pro-

tocol conformance. The work by Neykova et al. [81] extends preceding works with timed information: Scribble specifications are extended with clocks, resets, and clock predicates that constrain the occurrence of protocol interactions.

Recent work by Jia et al. [69] introduces a framework for monitoring interacting processes that follow binary session protocols, building upon a logically motivated theory of session types. As in several of the works mentioned above, in this framework monitors are placed next to communication endpoints.

A distinguishing aspect is blame assignment: in case processes deviate from the prescribed session protocols, monitors may halt the execution, raise an alarm, and assign blame. The authors prove that their dynamic monitoring is not intrusive in the sense that it does not change the behavior of well-typed processes. Also, they show that in case of alarm one of an indicated set of possible culprits must have been compromised.

Finally, we mention some works in which the concept of monitor as a process term, in the sense just described, has been exploited. Even though the main purpose of these works is not run-time verification, they can be seen as applications of choreographed monitoring. Di Giusto and Pérez [42], Di Giusto and Pérez [43] use this kind of monitors to support the run-time adaptation of session-typed processes in both binary and multiparty settings. There is exactly one monitor per session. By combining monitor information and event-based constructs, one may specify the reaction to unanticipated circumstances (for example, local failures) by means of adaptation steps. An associated type system ensures communication safety and consistency properties: while safety guarantees absence of run-time communication errors, consistency ensures that adaptation steps do not disrupt already established session protocols. In a similar line, the monitors defined by Castellani et al. [26,27] play a dual role: they enforce run-time adaptation policies, and ensure secure information flow in multiparty exchanges. Recent work by Mezzina and Pérez [75,76] uses monitors as the memories required to support models of concurrency in which actions are reversible and causally consistent.

6 Challenges and Conclusion

6.1 Challenges

We list here some challenges for future research in distributed runtime verification.

Fault Tolerance. One of the key characteristics of distributed systems is that, in practice, different parts of the system can fail independently. However, most approaches consider that the system does not fail. Some future problems include the following.

The theoretical approach by Fraigniaud et al. [53], Bonakdarpour et al. [19] (discussed in Sect. 5) has two major obstacles to become practical:

- First, after the distributed verdicts are emitted, there is a phase in which a global function is applied to the collection of verdicts emitted. This function must be implemented somehow by a central computational infrastructure which must receive all verdicts and produce an outcome. However, a general implementation of this function requires a non-failing central monitor. But the existence of such a central unit would greatly simplify the initial monitoring problem, and in fact, the basic starting point of [19,53] is to design distributed fault tolerant solutions.
- Second, the work [19] only presents an algorithm for the processing of a one letter observation, under the assumption that the processes are perfectly synchronised at the beginning of such an observation. To process a subsequent observation, the monitors that survive the first round must somehow re-synchronize, but again, a synchronisation procedure would provide a much simpler solution to the monitoring problem at hand. In summary, A general fault-tolerant solution for sequences of observations is still an open problem.

Also, there are very few results in runtime verification that can handle network failures (most notably, the work by Basin et al. [8]). It would be very interesting to extend these approaches to other logics and distributed system assumptions.

Global Atomic Observations. Specification formalisms for non-distributed systems assume that atomic predicates are testable, which is not a restriction. In distributed systems, in general, predicates are global in the sense that they can involve different parts of the system. Then, not all global predicates are Boolean combinations of local predicates. For example, one restriction of the work by Bauer and Falcone [11] is that the individual global observations are Boolean combinations of local observations performed in each of the processes, whose observations do not overlap. More formally, each process j can emit a collection of local propositions AP_j (such that $AP_j \cap AP_i = \emptyset$ whenever $i \neq j$). The alphabet of atomic observations is then $\Sigma_j = 2^{AP_j}$. Note how the global alphabet $\Sigma = 2^{\cup_i AP_i}$ is strictly larger, in general, than $\cup_i \Sigma_i$ because it can contain relational symbols like $p_i \vee p_j$ where p_i and p_j are local observations at different processes. We use *relational observations* to refer to atomic propositions whose truth value depends on the observations made at more than one process. For example, consider the numeric variables x_i and y_j where the sub-index indicates the process at which the variable is observed (P_i and P_j resp). The atomic predicate $x_i < y_j$ cannot be evaluated at P_i or P_j alone, and it cannot be decomposed into a Boolean combination of local predicates either.

As discussed earlier, even though research in predicate detection has considered classes of predicates richer than individual observations (regular, linear, etc.) and has characterised that detecting a predicate in 2-CNF is already NP-hard, it would be interesting to extend other techniques for decentralised and distributed monitoring beyond combinations of local predicates.

Monitor Orchestrations. Colombo and Falcone [34] present a choreographed decentralised monitoring solution obtained from a network of local monitors,

which is statically computed by mapping every subformula to a distributed system node. There are many possible ways to create such a network, even if one restricts the map (as in [34]) to one of the nodes with the highest number of propositions locally involved in the subformula, because there can be more than one such node. Even though all choices could lead to a correct monitoring solution, for a given trace of execution, the choice of network has an impact in the communication overhead. For every input trace, one could calculate a-posteriori the best network in the sense of the network that would have produced the lowest overhead. However, even for the fixed parameter assumed in [34] (e.g., static number of locations, fixed specifications, no dynamic remote spawning of new computation, the assumption of a global clock) it is not clear how to precompute an optimal network, even how to approximate it. Nevertheless, there are alternatives worth investigating. One plausible solution is to exercise the system in a test-bed to obtain input traces and compute the optimal network for the observed set of traces, with the assumption that the traces after deployment will involve similar communication flows. However, this kind of approach is not considered in [34].

Adequate solutions to this problem are probably even harder to come up with when proper distributed system constraints are considered, such as computation asynchrony, distributed clocks, and the possibility of partial failure. In practical settings, cases may even arise whereby one has to content with conflicting criteria. For instance, certain locations may not allow monitor processing and analysis to be carried out locally, forcing events to be communicated remotely to the analysing monitor. This, in turn, may conflict with confidentiality and security concerns.

Monitorability and Correctness. In general, the use of runtime analysis impinges on the extent to which a correctness property can be verified.

This aspect is often referred to as *monitorability*. One of the first works that introduces a notion of monitorability by defining classes of reactive languages that can be monitored is Viswanathan [91]. Later, D'Angelo et al. [40] defined monitorability for stream runtime verification on finite traces as the class of specifications for which efficient monitors can be generated. Pnueli and Zaks [84] formalised monitorability for LTL as the possibility of a finite trace to be extended to a finite witness of a specification satisfaction or violation. A similar notion was presented by Bauer et al. [13] and proved equivalent by Falcone et al. [48]. This notion was generalised to ω-regular languages by Falcone et al. [47] and Bauer [10], and later extended by Diekert et al. [44]. The tight complexity of this notion of monitorability was finally captured in [5]. An alternative definition of monitorability is given by Francalanza et al. [55] where the fragment of formulas of a given branching time logic that can be monitored at runtime is captured.

Decentralised and distributed monitoring introduces further restrictions and raises additional issues that may affect the monitorability of certain correctness properties. A first solution to decentralised monitorability was given recently

by El-Hokayem and Falcone [45], but further work will be necessary to study its full applicability and possible extensions.

Concurrent and distributed systems are notoriously hard to get right and these complications extend also to distributed runtime verification: errors arise only for particular sequences of events that are hard to simulate using pre-deployment techniques such as testing, and are also hard to trace and reproduce for analysis once they occur. It is thus imperative to continue to extend existing work on developing methods for ascertaining the correctness of the decentralised and distributed monitoring setups constructed along the lines of [16,49,54].

6.2 Conclusion

In this chapter we have surveyed the literature on runtime verification for distributed systems. After showing some practical motivations that have justified the study of monitoring techniques for distributed and decentralised systems, we identified a series of features that characterize and that allow to classify the different problems and approaches. These criteria include whether the solution involves or exploits the description of the system under analysis, whether there is a single central monitor or the monitoring task is distributed, whether there is an assumption on a global clock, and whether the system tolerates failures or perturbs the execution. Finally, we showed a comprehensive list of results proposed in the literature and listed some challenges for future work.

Acknowledgments. We are grateful to the anonymous reviewers for their useful remarks and suggestions, which led to significant improvements.

Financial Acknowledgements. This work was partially supported by COST Action IC1402 (Runtime Verification beyond Monitoring). César Sánchez is funded in part by Spanish MINECO Project "RISCO (TIN2015-71819-P)" and by EU H2020 project 731535 "Elastest". Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS – PEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

References

1. Alagar, S., Venkatesan, S.: Techniques to tackle state explosion in global predicate detection. IEEE Trans. Softw. Eng. (TSE) **27**(8), 704–714 (2001)
2. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016)
3. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 473–481. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_31
4. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. Wiley, Hoboken (2004)

5. Baader, F., Lippmann, M.: Runtime verification using the temporal description logic ALC-LTL revisited. J. Appl. Logic **12**(4), 584–613 (2014)
6. Bartocci, E.: Sampling-based decentralized monitoring for networked embedded systems. In: Bortolussi, L., Bujorianu, M.L., Pola, G. (eds.) Proceedings of the 3rd International Workshop on Hybrid Autonomous Systems (HAS 2013). EPTCS, vol. 124, pp. 85–99 (2013)
7. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring compliance policies over incomplete and disagreeing logs. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 151–167. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_17
8. Basin, D.A., Klaedtke, F., Zalinescu, E.: Failure-aware runtime verification of distributed systems. In: Proceedings of FSTTCS 2015, pp. 590–603 (2015)
9. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12. IEEE Computer Society (2006)
10. Bauer, A.K.: Monitorability of ω-regular languages. arXiv:1006.3638v1 (2010)
11. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 85–100. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_10
12. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. Formal Methods Syst. Des. **48**(1–2), 49–93 (2016)
13. Bauer, A.K., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**, 14:1–14:64 (2011)
14. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime verification with minimal intrusion through parallelism. Formal Methods Syst. Des. **46**(3), 317–348 (2015)
15. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Proceedings of FMOODS/FORTE 2013, pp. 50–65 (2013)
16. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
17. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: Automated conflict-free distributed implementation of component-based models. In: Proceedings of the IEEE 5th International Symposium on Industrial Embedded Systems (SIES 2010), pp. 108–117. IEEE (2010)
18. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: Proceedings of EMSOFT 2010, pp. 209–218. ACM (2010)
19. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D., Travers, C.: Decentralised asynchronous crash-resilient runtime verification. In: Proceedings of the 27th International Conference on Concurrency Theory (CONCUR 2016). LIPIcs, vol. 59, pp. 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
20. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Travers, C.: Challenges in fault-tolerant distributed runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part II. LNCS, vol. 9953, pp. 363–370. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_27
21. Carbone, M. (ed.): Proceedings of the Third Workshop on Behavioural Types (BEAT 2014). EPTCS, vol. 162 (2014)

22. Cassar, I., Francalanza, A.: On synchronous and asynchronous monitor instrumentation for actor systems. In: Proceedings of FOCLASA 2014, vol. 175, pp. 54–68 (2014)
23. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 38–54. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_3
24. Cassar, I., Francalanza, A.: On implementing a monitor-oriented programming framework for actor systems. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 176–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_12
25. Cassar, I., Francalanza, A., Said, S.: Improving runtime overheads for detector. In: Buhnova, B., Happe, L., Kofron, J. (eds.) Proceedings of the 12th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2015). EPTCS, vol. 178, pp. 1–8 (2015)
26. Castellani, I., Dezani-Ciancaglini, M., Pérez, J.A.: Self-adaptation and secure information flow in multiparty structured communications: a unified perspective. In: [21], pp. 9–18
27. Castellani, I., Dezani-Ciancaglini, M., Pérez, J.A.: Self-adaptation and secure information flow in multiparty communications. Formal Asp. Comput. **28**(4), 669–696 (2016)
28. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
29. Chase, C.M., Garg, V.K.: Detection of global predicates: techniques and their limitations. Distrib. Comput. **11**(4), 191–201 (1998)
30. Chauhan, H., Garg, V.K., Natarajan, A., Mittal, N.: A distributed abstraction algorithm for online predicate detection. In: IEEE 32nd Symposium on Reliable Distributed Systems (SRDS 2013), pp. 101–110. IEEE Computer Society (2013)
31. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
32. Colombo, C., Dimech, G., Francalanza, A.: Investigating instrumentation techniques for ESB runtime verification. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 99–107. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_7
33. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 140–155. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_12
34. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods Syst. Des. **49**(1–2), 109–158 (2016)
35. Colombo, C., Francalanza, A., Gatt, R.: Elarva: a monitoring tool for Erlang. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 370–374. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_29
36. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: polyLarva: runtime verification with configurable resource-aware monitoring boundaries. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 218–232. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_15
37. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 163–173 (1991)

38. Coulouris, G.: Distributed Systems: Concepts and Design. Addison-Wesley, Boston (2011)
39. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Trans. Parallel Distrib. Syst. **10**(6), 642–657 (1999)
40. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th International Symposium of Temporal Representation and Reasoning (TIME 2005), pp. 166–174. IEEE CS Press (2005)
41. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Formal Methods Syst. Des. **46**(3), 197–225 (2015)
42. Di Giusto, C., Pérez, J.A.: An event-based approach to runtime adaptation in communication-centric systems. In: Hildebrandt, T., Ravara, A., van der Werf, J.M., Weidlich, M. (eds.) WS-FM 2014-2015. LNCS, vol. 9421, pp. 67–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33612-1_5
43. Di Giusto, C., Pérez, J.A.: Event-based run-time adaptation in communication-centric systems. Formal Aspects Comput. **28**(4), 1–36 (2016)
44. Diekert, V., Muscholl, A., Walukiewicz, I.: A note on monitors and Büchi automata. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 39–57. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_3
45. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017), pp. 125–135. ACM (2017)
46. Falcone, Y., Cornebize, T., Fernandez, J.-C.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 66–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43613-4_5
47. Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_4
48. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? STTT **14**(3), 349–382 (2012)
49. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Softw. Syst. Model. **14**(1), 173–199 (2015)
50. Felser, M.: Real-time Ethernet - industry prospective. Proc. IEEE **93**(6), 1118–1129 (2005)
51. Fidge, C.: Timestamps in message-passing systems that preserve the partial ordering. In: Proceedings of the 11th Australian Computer Science Conference, pp. 55–66 (1989)
52. Fidge, C.: Logical time in distributed computer systems. Computer **24**(8), 28–33 (1991)
53. Fraigniaud, P., Rajsbaum, S., Travers, C.: On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In: RV (2014)
54. Francalanza, A.: A theory of monitors. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 145–161. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_9
55. Francalanza, A., Aceto, L., Ingolfsdottir, A.: Monitorability for the Hennessy-Milner logic with recursion. FMSD **51**(1), 1–30 (2017)

56. Francalanza, A., Gauci, A., Pace, G.J.: Distributed system contract monitoring. In: Pimentel, E., Valero, V. (eds.) Proceedings of the Fifth Workshop on Formal Languages and Analysis of Contract-Oriented Software, FLACOS 2011, Málaga, Spain, 22–23 September 2011. EPTCS, vol. 68, pp. 23–37 (2011)
57. Francalanza, A., Gauci, A., Pace, G.J.: Distributed system contract monitoring. J. Logic Algebraic Program. **82**(5–7), 186–215 (2013)
58. Francalanza, A., Hennessy, M.: A theory of system behaviour in the presence of node and link failure. Inf. Comput. **206**(6), 711–759 (2008)
59. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 112–129. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_7
60. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. FMSD **46**(3), 226–261 (2015)
61. Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: theory, implementation and applications. In: Proceedings of CCS 2011, pp. 151–162 (2011)
62. Garg, V.K.: Elements of Distributed Computing. Wiley-IEEE Press, New York (2002)
63. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001), pp. 322–329. IEEE CS Press (2001)
64. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **5**(3), 299–307 (1994)
65. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
66. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
67. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations - distributed dynamic verification with session types and python. In: [72], pp. 130–148
68. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
69. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 582–594. ACM (2016)
70. Joyce, J., Lomow, G., Slind, K., Unger, B.W.: Monitoring distributed systems. ACM Trans. Comput. Syst. **5**(2), 121–150 (1987)
71. Lamport, L.: Time, clocks and the ordering of events in distributed systems. Commun. ACM **21**(7), 558–565 (1978)
72. Legay, A., Bensalem, S. (eds.): RV 2013. LNCS, vol. 8174. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1
73. Luo, Q., Roşu, G.: EnforceMOP: a runtime property enforcement system for multithreaded programs. In: ISSTA. ACM, New York (2013)
74. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215–226. Elsevier (1989)

75. Mezzina, C.A., Pérez, J.A.: Reversible sessions using monitors. In: Orchard, D.A., Yoshida, N. (eds.) Proceedings of the Ninth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES 2016). EPTCS, vol. 211, pp. 56–64 (2016)
76. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebraic Methods Program. **90**, 2–30 (2017)
77. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992)
78. Mittal, N., Garg, V.K.: Techniques and applications of computation slicing. Distrib. Comput. **17**(3), 251–277 (2005)
79. Mittal, N., Sen, A., Garg, V.K.: Solving computation slicing using predicate detection. IEEE Trans. Parallel Distrib. Systems (TPDS) **18**(12), 1700–1713 (2007)
80. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015), pp. 494–503. IEEE Computer Society (2015)
81. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. In: [21], pp. 19–26
82. Neykova, R., Yoshida, N., Hu, R.: SPY: local verification of global protocols. In: [72], pp. 358–363
83. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 420–434. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75142-7_32
84. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_38
85. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005)
86. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, pp. 171–183. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27860-3_17
87. Sen, A., Garg, V.K.: Partial order trace analyzer (POTA) for distributed programs. ENTCS **89**(2), 22–43 (2003). Proceedings of Workshop on Runtime Verification (RV 2003)
88. Sen, A., Garg, V.K.: Formal verification of simulation traces using computation slicing. IEEE Trans. Comput. **56**, 511–527 (2007)
89. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of ICSE 2004. IEEE CS Press (2004)
90. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Decentralized runtime analysis of multithreaded applications. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006). IEEE (2006)
91. Viswanathan, M.: Foundations for the run-time analysis of software systems. Ph.D. thesis, University of Pennsylvania (2000)
92. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3
93. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: combining synchronous and asynchronous monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 482–490. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_32