International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-023-00713-2

GENERAL

Special Issue: RV 2020

Concurrent runtime verification of data rich events

Nastaran Shafiei[1] · Klaus Havelund[2] · Peter Mehlitz[1]

Accepted: 30 May 2023

Abstract
This paper presents the open-source runtime verification tool MESA (MEssage-based System Analysis), implemented in Scala, which supports concurrent monitors using the Actor model. Furthermore, the tool supports indexing (slicing) on the data values occurring in data-carrying events, for each individual monitor. The tool is generic in the sense that any monitoring system can be used for creating monitors. In this paper, we use the internal Scala DSL Daut for programming such data-parameterized state machines and temporal logic. To illustrate MESA/Daut, we present a case study that monitors flights from live U.S. airspace data streams, verifying that they conform to planned routes. With base in the case study, we then perform an extensive empirical study of the potential benefits from monitoring slices of a single property in concurrently executing actors. Due to the overhead of scheduling "small" actors (one for each slice or a small number of slices), it is not obvious that concurrent execution of such is beneficial. However, as a main result, we demonstrate that concurrent monitoring of slices to handle data-carrying events can provide considerable speed gains.

Keywords Runtime verification · First-order temporal properties · Slicing · Concurrency · Actors · Scala

1 Introduction

Distributed computing is becoming increasingly important as almost all modern systems in use are distributed. Distributed systems usually refer to systems with components that are placed at different locations, and that communicate via message passing. These systems are known to be very hard to reason about due to certain characteristics, e.g., their concurrent nature, non-determinism, and communication delays [24, 36]. There has been a wide variety of work focusing on verifying distributed systems, including dynamic verification techniques such as runtime verification [20, 39] which checks if a run of a System Under Observation (SUO) satisfies properties of interest. Properties are typically captured as formal specifications expressed in forms of linear temporal logic, finite state machines, regular expressions, etc. Some of the proposed runtime verification techniques related to distributed computing employ concurrent monitoring [8, 10, 13, 16, 23, 26]. Using concurrency, one can benefit from parallel execution of the concurrent units, which can improve the overall performance. One can exploit parallelism by using additional hardware resources for running monitors to reduce their online overhead [13]. Using concurrent monitors instead of one monolithic monitor, one can achieve higher utilization of available cores [23].

In this paper, we propose a concurrent runtime verification approach for analyzing distributed systems. Note that our runtime verification approach can itself be distributed, however, the distributed setting is not demonstrated in this paper. Our approach is not tied to a particular SUO, although it is motivated by a use case which aims to analyze flight behaviors in the *National Airspace System* (NAS). We pursue this use case as our case study. NAS refers to the U.S. airspace and all of its associated components, including airports, airlines, air navigation facilities, services, rules, regulations, procedures, and workforce. NAS is a highly distributed and large system with over 19,000 airports, including public, private, and military airports, and up to 5,000 flights in the U.S. airspace at the peak traffic time. NAS actively evolves under the NextGen (Next Generation Air Transportation System) project, led by the Federal Aviation Administration (FAA), which aims to modernize NAS by introducing new concepts,

✉ N. Shafiei
nastaran.shafiei@nasa.gov

K. Havelund
klaus.havelund@jpl.nasa.gov

P. Mehlitz
peter.c.mehlitz@nasa.gov

[1] NASA Ames Research Center/KBR Inc., Moffett Field, CA 94035, USA

[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, USA

Published online: 26 June 2023

and technologies. Considering the size and complexity of NAS, efficiency is vital to our approach. Our ultimate goal is to generate a monitoring system that handles high volumes of live data feeds and can be used as a ground control station to analyze air traffic data in NAS.

Our approach is based on employing concurrent monitors, and adopts *the actor programming model* for building concurrent systems. The actor model was proposed in 1973 as a way to deal with concurrency in high performance systems [31]. Concurrent programming with shared memory is notoriously difficult, creating problems such as synchronization and memory protection. The actor programming model offers an alternative that is supposed to eliminate such problems. The primary building blocks in the actor programming model are *actors*, which are concurrent objects that do not share states and only communicate by means of asynchronous messages that do not block the sender. Actors are fully independent and autonomous and only become runnable when they receive a message in their buffer, called the *mailbox*. The model also guarantees that each runnable actor only executes in one thread at a time, a property which allows for viewing an actor's code as a sequential program.

Our approach is implemented as the framework MESA, which is an open source project [43]. MESA uses the Akka toolkit [4, 51], that provides an implementation of the actor model in Scala. The actor model is adopted by numerous frameworks and libraries. However, what makes Akka special is how it provides support and additional tooling for building actor-based systems. MESA also leverages the Runtime for Airspace Concept Evaluation (RACE) [40, 41] framework, another system built on top of Akka and extending it with additional features. RACE is a framework to generate airspace simulations, and provides actors to import, translate, filter, archive, replay, and visualize data from NAS, that can be directly employed in MESA when checking for properties in the NAS domain.

MESA supports specification of properties in data-parameterized temporal logic and state machines. The support for formal specification is provided by integrating the trace analysis tools TraceContract [9, 30] and Daut (Data automata) [28, 29], implemented as domain-specific languages (DSLs) [5]. TraceContract, which was also used for command sequence verification in NASA's LADEE (Lunar Atmosphere And Dust Environment Explorer) mission [37], supports a notation that combines data-parameterized state machines, referred to as data automata, with temporal logic. Daut is a modification of TraceContract, which, amongst other things, allows for more efficient monitoring. In contrast to general-purpose languages, *external* DSLs offer high levels of abstractions but usually limited expressiveness. TraceContract and Daut are, in contrast, *internal* DSLs since they are embedded in an existing language, Scala, rather than providing their own syntax and runtime support. Thus, their specification languages offer all features of Scala, which adds adaptability and richness.

As a basic optimization technique, MESA supports indexing, a restricted form of slicing [42, 48]. Indexing slices the trace up into sub-traces according to selected data in the trace, and feeds each resulting sub-trace to its own sub-monitor. As an additional optimization technique, MESA allows concurrency at three levels. First, MESA runs in parallel with the monitored system(s). Second, multiple properties are translated to multiple concurrently running monitors, one for each property. Third, and most importantly for this presentation, each property is checked by multiple concurrent monitors by slicing the trace up into sub-traces using indexing, and feeding each sub-trace to its own concurrent sub-monitor.

MESA is highly configurable. One can configure MESA by specifying how to check a property using concurrent monitors. By tuning relevant parameters in the MESA configuration, and evaluating the performance, one can find the optimal number of concurrent monitors to improve the performance. As the main result of the paper, we demonstrate that concurrent execution of slices (the third form of concurrency mentioned above) is beneficial with respect to performance. This is not an obvious result considering the cost of scheduling threads for small tasks.

Note that this paper is an extended and revised version of a preliminary conference paper, which was presented in RV 2020 [52]. The present paper elaborates more on the implementation of the tool, and the case study. It includes the description of the underlying threading model and how it is set up in our experiments. Finally, it expands on related work, and the results obtained from our experiments.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 provides an overview of the MESA architecture. Section 4 presents the case study illustrating how MESA can be used to monitor arrivals to US airports. Section 5 gives a brief introduction to the Akka threading model in Scala, which is useful for understanding the following experiment. Section 6 presents an experiment rooted in the case study, analyzing the combination of concurrency and indexing, a main contribution of the paper. Section 7 discusses the overall approach and results. Finally, Sect. 8 concludes the paper.

2 Related work

Amongst the most relevant work is that of Hallé et al. [26]. In this work, the authors use data parallelism to scale first-order temporal logic monitoring by slicing the trace into multiple sub-traces, and feeding these sub-traces to different parallel executing monitors. The approach creates as many monitors as there are slices. The individual monitors are considered black boxes, which can host any monitoring system fitting

the expected monitor interface. The authors do not provide performance results for specifically the combination of slicing and concurrency. Their approach is implemented as an extension of BeepBeep 3 [25] which allows certain types of computations to be performed in parallel. However, the parallelism in BeepBeep 3 is done manually, which requires much fine-tuning.

Another attempt in a similar direction is that of Basin et al. [10] which also submits trace slices to parallel monitors, a development of the author's previous work on using MapReduce for the same problem [8]. The authors provide performance results for the use of slicing together with concurrency, but do not compare these with runs without concurrency. However, their evaluation shows that their monitoring approach scales to large logs. The logs analyzed contain billions of events, supporting the observation that exactly this use of concurrency is performance enhancing. The approach is proposed for offline verification where traces are logged in a distributed file system.

Reger in his MSc dissertation [47] experimented with creating parallel monitors to monitor subsets of the state space for each submitted event. However, in that early work, the results were not promising as using concurrency slowed down the monitoring process, possibly due to the less mature state of support for parallelism in Java and hardware at the time. As Reger writes in [47] (page 81): *"Monitoring the DaCapo benchmarks gave barely any good results ... the concurrency offered by a multicore system was insignificant and these approaches often made the benchmarks run much slower than with just the base monitor"*. Reger also later in [49] (page 2) writes: *"In previous work [47], we attempted to parallelise the way that a RuleR monitor handles events and found that the amount of work required to evaluate a single event was generally not large enough to benefit from parallelisation. For runtime monitoring as usually envisaged, with simple events and event processing, we believe this to be generally the case when considering a single step of a monitor."*.

Berkovich et al. [13] also address the splitting of the trace according to data into parallel executing monitors. However, differently from the other approaches, the monitors run on GPUs instead of on CPUs, as the system being monitored does. Their monitoring approach incurs minimal intrusion, as the execution of monitoring tasks takes place on different computing hardware than the execution of the SUO.

Francalanza and Seychell [23] explore structural parallelism, where parallel monitors are spawned based on the structure of the formula. E.g., a formula $p \wedge q$ will cause two parallel monitors, one for each conjunct, co-operating to produce the combined result.

El-Hokayem and Falcone [18] review different approaches to monitoring multithreaded Java programs, which differs in perspective from the monitoring system itself to be parallel. Francalanza et al. [24] survey runtime verification research on how to monitor systems with distributed characteristics, solutions that use a distributed platform for performing the monitoring task, and foundations for decomposing monitors and expressing specifications amenable for distributed systems.

In [12], Basin et al. present an approach to scaling monitoring of distributed systems. Their approach assumes that the monitoring system receives more than one input stream, and that events can arrive out of order. Note that in the here presented work we also target monitoring of distributed systems, however, in our case the monitoring system is dealing with only a single input stream.

The work by Burlò et al. [14] targets open distributed systems and relies on session types for specification of communication protocols. It applies a hybrid verification technique where the components available pre-deployments are checked statically, and the ones that become available at runtime are verified dynamically. Their approach is based on describing communication protocols via session types with assertions, from the `lchannels` Scala library, which are used to synthesize monitors automatically. They develop a formal model of processes monitored using session types, and prove the correctness of their approach in terms of soundness and completeness. Moreover, they show the feasibility of their approach through a set of benchmarks.

The work by Neykova and Yoshida [45] applies runtime verification to ensure a *sound* recovery of distributed Erlang processes after a failure occurs. Their approach is also based on session types to enforce protocol conformance.

The work by Attard and Francalanza [6] targets asynchronous distributed systems. Their approach allows for generating partitioned traces at the instrumentation level, where each partitioned trace provides a localized view for a subset of the SUO. The work focuses on global properties that can be cleanly decomposed into a set of local properties, which can be verified against local components. It is suggested that one could use the partitioned traces to infer alternative merged execution traces of the system. The implementation of the approach targets actor-based Erlang systems, and includes concurrent localized monitors captured by Erlang actors.

The work by El-Hokayem and Falcone [19] targets decentralized systems that consist of multiple components without a central observation point. They present a general algorithm to monitor decentralized specifications which are composed of a set of automata captured by monitors attached to components. They also elaborate on two properties of decentralized specifications, monitorability and compatibility. The former ensures that the monitors are able to reach a verdict for all possible traces, and the later ensures that a specification can be deployed on a given architecture.

In [38], Lavery et al. present an actor-based monitoring framework in Scala, that, similar to our approach, is built

using the Akka toolkit. The monitoring system does not, in contrast to our approach, provide a temporal logic API for specifying properties, which is argued to be an advantage. Note, that Daut as well as TraceContract, in addition to such a temporal logic API, allow defining monitors using any Scala code as well. A monitor *master* actor can submit monitoring tasks to *worker* actors in an automated round-robin fashion. This, however, requires that the worker monitors do not rely on an internal state representing a summary of past events.

The work by Aceto et al. [2] presents a synthetic benchmarking framework for evaluating runtime verification tools that can target concurrent message-based systems. The benchmarks are synthesised as Erlang actor-based systems. The work performs an empirical study, which reports on overhead from synthesised monitors that are inlined into the system.

Monitoring of hyperproperties [3, 21] is a more recent research topic, where a property is a set of sets of traces (instead of a set of traces). In a hyperproperty temporal logic, a formula can relate multiple executions of a, not necessarily distributed, system to each other. The concept of hyperproperties was initially suggested in [15] as a means to express security policies that cannot be expressed as traditional single-trace properties. A yet unexplored question is whether MESA can be used for monitoring hyperproperties.

Two high-performance systems in particular paved the way wrt. slicing, first MOP [42] and later MarQ [48]. Our approach to slicing can be compared to those efforts by focusing on the expressiveness of the slicing along two dimensions, which we shall call *dispersed slicing* and *partial slicing*. Both MOP and MarQ support *dispersed slicing* by allowing event parameters used for indexing to arrive in different events. In contrast, in our approach such event parameters have to all arrive in the same event. The classic example is the unsafe map iterator property [42], which states that iteration over the keys (domain) of a map in Java is safe. Specifically, if an observed event reports that the keys of a map m are extracted as a collection c, and a next event reports that an iterator i is extracted from the collection c, and a third event reports that the map m is updated, then after that it is unsafe to continue iterating over the iterator i. In this case, the keys m and c are introduced in the first event and i is introduced in the second event. Note, that even though we do not support dispersed indexing, properties like this can be expressed, although with less efficient execution.

Both MarQ and our approach support *partial slicing* by allowing a strict subset of the parameters to an event to be used for indexing. In contrast, MOP requires all event parameters to be used for indexing. This limits the expressiveness of MOP. Say for example that we monitor acquisitions and releases of locks l by threads t in a concurrent system via events of the form $acq(t,l)$ and $rel(t,l)$. A property that cannot be expressed in MOP is that if a lock l has been acquired by a thread t, indicated by the event $acq(t,l)$, then another thread t' cannot acquire the lock until t has released it. The reason is that $acq(t,l)$ and $acq(t',l)$ will be sent to two different slices.

3 An overview of MESA

Our approach implemented in MESA allows for concurrent monitoring of formal properties. As shown in this section, MESA is designed in a way to ensure concurrency at three different levels. It can run in parallel with the SUO. Moreover, multiple properties can be captured by multiple concurrent monitors. Finally, one property can be translated into multiple concurrent monitors. Furthermore, MESA is designed to provide flexibility in terms of the system used for property specification. We elaborate on this by explaining how the Daut system is integrated into MESA.

MESA is a framework for building actor-based monitoring systems. An overview of a system that can be built using MESA is shown in Fig. 1. A MESA system is solely composed of actors that implement a pipeline of four processing steps. The vertical lines between actors represent publish-subscribe communication channels, resembling pipelines where outputs from one step are used as inputs for the following step. The first step is *data acquisition*, which extracts data from the SUO. The second step is *data processing*, which parses raw data extracted by the previous step and generates a trace composed of events that are relevant to the properties of interest. Next step is *monitoring* which checks the trace obtained from the previous step against the given properties. Finally, the last step is *reporting* which presents the verification results. What MESA offers are the building blocks to create actors for each step of the runtime verification. Often one needs to create application specific actors to extend MESA towards a particular domain. Besides the NAS domain, MESA is extended towards the UxAS project, which is developed at Air Force Research Laboratory and provides autonomous capabilities for unmanned systems [46].

Akka actors can use a point-to-point or publish-subscribe model to communicate with one another. In point-to-point messaging, the sender sends a message directly to the receiver, whereas, in publish-subscribe messaging, the receivers subscribe to a channel, and messages published on that channel are forwarded to them by the channel. Messages sent to each actor are placed on its mailbox. Only actors with a non-empty mailbox become runnable. Actors extend the `Actor` base trait and implement a method `receiveLive` of type `PartialFunction[Any, Unit]` which captures their core behavior. It includes a list of `case` statements that, by applying Scala pattern matching over parameterized events, determine the messages that can be handled by the actor and the way they are processed. To create a MESA monitor-

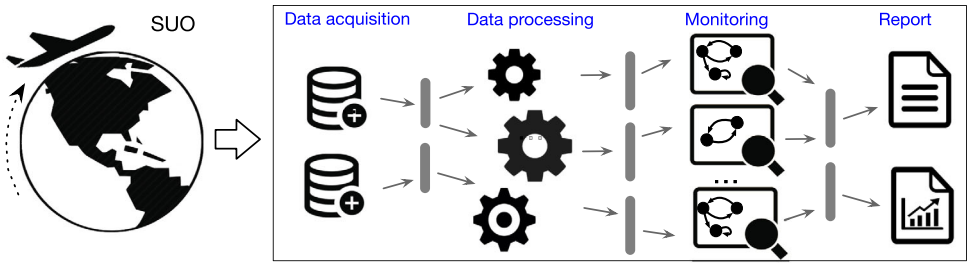


Fig. 1 Overview of a MESA actor-based monitoring system

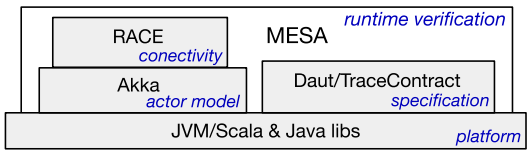


Fig. 2 The MESA framework infrastructure

ing system (Fig. 1), one needs to specify the actors and the way they are connected with communication channels in a HOCON [32] configuration file used as an input to MESA.

Figure 2 shows the MESA framework infrastructure and the existing systems incorporated into MESA. These systems are all open source Scala projects. MESA is also written in Scala, and it is open source, available at [43]. Akka provides the actor model implementation. RACE, built on top of Akka, is used for connectivity to external systems. MESA employs a non-intrusive approach since for safety-critical systems such as NAS, sources are either not available or are not allowed to be modified for security and reliability reasons. Even when the source is available, any potential malfunction that may be introduced by instrumentation cannot be tolerated. RACE provides dedicated actors, referred to as importers, that can subscribe to commonly-used messaging system constructs, such as JMS server and Kafka. Using an importer actor from RACE in the data acquisition step, we extract data from the SUO, in a nonintrusive manner. Moreover, RACE extends Akka with features that our framework relies on, such as synchronizing the execution of actor lifetime phases including instantiation, initialization, start, and termination. RACE also provides a mechanism to let remote actors communicate with local actors seamlessly using the same API. By incorporating remote actors, one can create a distributed monitoring system using MESA, where actors are placed on different machines.

MESA incorporates the tools TraceContract [9, 30] and Daut [28, 29] for property specification. TraceContract and Daut are both internal (embedded) trace analysis DSLs (Scala libraries), where given a program trace and a formalized property, they determine whether the property holds for the trace. Since they are internal DSLs, they allow for the use of full Scala for writing monitors, in addition to, and in combination with, using the DSLs. This allows for very expressive monitors, which can not only perform arbitrary data computations, but also give rich verdicts beyond the Boolean domain, such as results of such computations. `Monitor` is the main class in these DSLs (`tracecontract.Monitor` and `daut.Monitor`), which encapsulates property specification capabilities. It implements the method `verify`, that for each incoming event updates the state of the monitor accordingly.

Properties are defined as subclasses of class `Monitor`, and instances of such are referred to as monitors from here on. Similar to the actor `receiveLive` method, a user-defined subclass of the class `Monitor` includes a series of `case` statements that determine the events that can be handled by the monitor and the behavior triggered for each event. The properties described in this paper are specified using Daut since it also provides an indexing capability within monitors to improve their performance. It allows for defining a function from events to keys, where keys are used as entries in a hash map to obtain those states which are relevant to an event. Using indexing, a Daut monitor, when receiving an event, only iterates over an indexed subset of states relevant for the event instead of the entire set, yielding a performance improvement.

The actors in the monitoring step (Fig. 1), referred to as *monitor actors*, hold instances of the `Monitor` classes and feed them with incoming event messages. MESA provides components referred to as *dispatchers* which are configurable and can be used in the monitoring step to determine how the check for a property is distributed among different monitor actors. Dispatchers, implemented as actors, can generate monitor actors on-the-fly and distribute the incoming trace between the monitor actors, relying on identifiers extracted from data parametrized events. Dispatchers are key to our experiments.

The UML diagram in Fig. 3 illustrates how Daut is integrated into MESA. The integration of TraceContract is performed in a similar manner. The integration ensures that code is not tied to any specific trace analysis DSL to provide extensibility when employing new DSLs. To integrate a trace analysis DSL, MESA includes a class that extends the key class in DSL that provides property specification capabilities (e.g., `daut.Monitor` in Daut). The class used

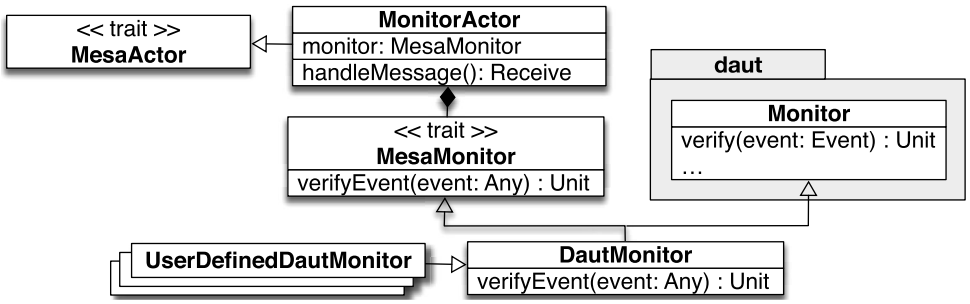


Fig. 3 Integration of the Daut DSL into the MESA framework

to integrate the DSL also implements a trait which is called `MesaMonitor` and used as a Scala mixin.[1]

To integrate Daut, MESA implements the class `DautMonitor`, which extends `daut.Monitor` and implements `MesaMonitor`. The `MesaMonitor` mixin is used to establish a common interface throughout the code to refer to all monitor objects. It defines the `verifyEvent` method, which is implemented by subtypes and checks the incoming events against the specified properties by delegating the verification to the DSL code (e.g. `daut.Monitor.verify(event: Event)` for a Daut monitor). Monitor actors in the monitoring phase extend the class `MonitorActor`. This class has a field of type `MesaMonitor`, which is set to a monitor object during the actor initialization. The concrete type for the monitor object is specified in the monitor actor's configuration. For each incoming event placed in the monitor actor mailbox, the monitor actor invokes the method `verifyEvent` on its underlying monitor object to verify the event against the properties implemented by the monitor.

4 Monitoring live flights in the U.S. airspace

This section presents the case study where MESA is applied to check a property known as RNAV STAR adherence, referred to as P_{RSA} in this paper. RNAV (Area Navigation) [56] is a navigation system that allows the aircraft to move on any desired flight route, provided as a sequence of waypoints, without relying on ground-based navigation aids. The RNAV system is based on instrument flight rules [57] which are a set of regulations under which the aircraft is navigated only by reference to the instruments in the aircraft cockpit rather than using visual references. RNAV systems continuously determine the position of the aircraft and by providing deviation from the desired route, they aid the pilot to navigate the aircraft.

A STAR is a standard arrival procedure designed by the FAA to transition flights from the en-route phase to the approach phase, where descent starts. Every STAR specifies a set of flight routes (See Fig. 4), where each route is specified by a sequence of *waypoints*, accompanied by vertical and speed profiles specifying altitude and airspeed restrictions. A waypoint is a geographical position with latitude and longitude coordinates. A STAR is a form of communication between the flight crew and air traffic controllers. When the air traffic controller gives a clearance to the pilot to take a certain STAR route, they communicate the route, altitude, and airspeed. A STAR route, assigned to a flight, is encoded in the flight plan, presented to the pilot as a sequence of waypoints. Certain STARs, that can be only used by aircraft equipped with RNAV navigation systems, are referred to as RNAV STARs. One of the ongoing focus points of the FAA is to increase the utilization of RNAV-based procedures, which reduce the communication overhead with ground. Figure 4 demonstrates a RNAV STAR procedure, called BDEGA3, which is designed for the SFO airport. There are a total of 10 RNAV STAR procedures assigned to the SFO airport.

From 2009 to 2016, as part of the NextGen project, 264 more RNAV STAR procedures were implemented on an expedited timeline [56] which led to safety concerns raised by airlines and air traffic controllers, including numerous unintentional pilot deviations [17, 33]. A possible risk associated with deviating from a procedure is a loss of separation, which can result in a midair collision. The work presented in [54] studies RNAV STAR adherence trends based on a data mining methodology, and shows deviation patterns at major airports [7].

The case study applies runtime verification to check if flights are compliant with the designated RNAV STAR routes in real-time. A navigation specification for flights assigned to a RNAV STAR requires a lateral navigation accuracy of 1 NM[2] for at least 95% of the flight time [34]. Our approach focuses on lateral adherence, where incorporating a check for vertical and speed profiles becomes trivial. We informally define the RNAV STAR lateral adherence property as follows, adopted by others [54].

P_{RSA}: *a flight shall cross inside a 1.0 NM radius around each waypoint in the assigned RNAV STAR route, in order.*

[1] "Mixing in" traits (mixins) is a way of allowing for a class (or trait) to extend multiple traits. These are included, rather than inherited from, avoiding the problems of multiple inheritance.

[2] NM, nautical mile is a unit of measurement equal to 1,852 meters.

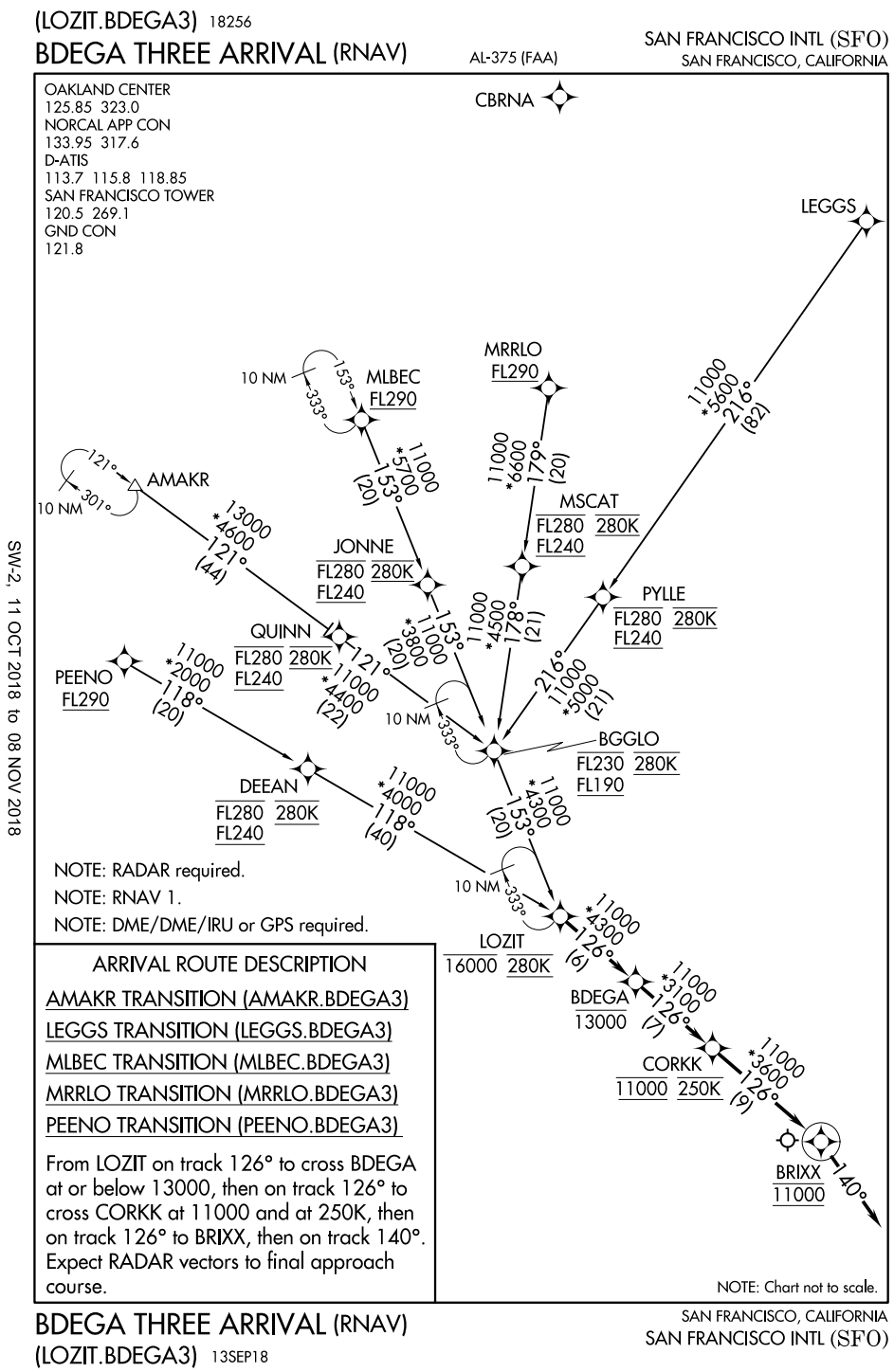


Fig. 4 BDEGA3 RNAV STAR procedure designed for SFO

4.1 Formalizing property P_{RSA}

For the sake of brevity, we say that a flight *visits* a waypoint if the flight crosses inside a 1.0 NM radius around the waypoint. We say that an event occurs when the aircraft under scrutiny visits a waypoint that belongs to its designated RNAV STAR route. For example, in Fig. 5, where circles represent 1.0 NM radius around the waypoints, the sequence of events for this aircraft is MLBEC MLBEC JONNE.

We define a state machine capturing Property P_{RSA}. Let L be a set including the labels of all waypoints in the RNAV STAR route. Let *first* and *last* be predicates on L that denote the initial and final waypoints, respectively. Let *next* be a partial function, $L \hookrightarrow L$, where given a non-final waypoint

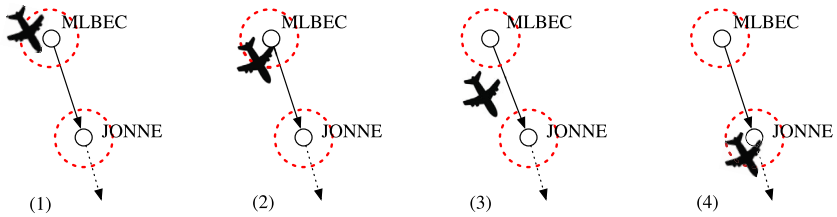


Fig. 5 The sequence of events for the aircraft is MLBEC MLBEC JONNE

in L it returns the subsequent waypoint in the route. For example, *next*(MLBEC) returns JONNE (Fig. 5). The finite state machine for Property P_{RSA} is the tuple $(Q, \Sigma, q_0, F, \delta)$ where

- $Q = L \cup \{init, err, drop\}$
- $\Sigma = \{e_t \mid t \in L \cup \{FC, SC\}\}$
- $q_0 = init$
- $F = \{err, drop\} \cup \{q \in L \mid last(q)\}$
- $\delta : Q \times \Sigma \rightarrow Q$

Q is the set of all states, and *init* is the initial state. Σ is the set of all possible events. The event e_t where $t \in L$ indicates that the aircraft visits the waypoint t. The event e_{FC} indicates that the flight is completed, and e_{SC} indicates that the flight is assigned to a new STAR route. Note that *FC* stands for flight completed and *SC* stands for STAR changed. F is the set of final states, where *last* represents the set of accept states indicating that the flight adhered to the assigned RNAV STAR route. The state *err* represents an error state indicating the violation of the property. The state *drop* represents a state at which the verification is dismissed due to assignment of a new STAR route. The transition function δ is defined as below.

$$\delta(q, e_t) = \begin{cases} t & if (q = init \ \& \ first(t)) \\ & \quad \text{or } (q \in \{x \in L \mid \neg last(x)\} \ \& \\ & \quad\quad t \in \{q, next(q)\}) \\ err & if (q = init \ \& \ t \neq SC \ \& \ \neg first(t)) \\ & \quad \text{or } (q \in \{x \in L \mid \neg last(x)\} \ \& \\ & \quad\quad t \notin \{q, next(q), SC\}) \\ drop & if (q \neq err \ \& \ t = SC) \end{cases}$$

At *init*, if the flight visits the first waypoint of the assigned route, the state machine advances to the state representing the first waypoint. Alternatively, if at waypoint q, the flight can only visit q or the next waypoint in the route, $next(q)$. Otherwise, if at *init*, and it visits any waypoint other than the first waypoint of the route, the state machine advances to *err*. Likewise, if the flight visits any waypoint not on the route, the state advances to *err*. Finally, at any state other than *err*, if the flight gets assigned to a new route ($t = SC$), the state machine advances to *drop*.

4.2 P_{RSA} monitor implementation

Event types are implemented as Scala `case` classes due to their concise syntax and built-in pattern matching support, that facilitates convenient programming of transitions between states. There are three such event types.

```
case class Visit(info: Info, wp: Waypoint)
case class Completed(track: Track)
case class StarChanged(track: Track)
```

The class `Visit` represents an event indicating that a flight visits a given waypoint `wp` of type `Waypoint`. The `info` argument of type `Info` carries information about the flight, including its state and its track. The state captures position, heading, speed, etc. and the track captures the assigned STAR route. Each flight is uniquely identified by a so-called call sign (`cs`), of type `String`, which is part of the state as well as of the track. The class `Completed` represents an event indicating that a flight is completed. Finally, the class `StarChanged` represents an event indicating that a flight is assigned a new STAR route.

We implement the property P_{RSA} as the Daut monitor in Fig. 6. A Daut monitor is defined as a class, in this case `P_RSA`, which, directly or indirectly, extends the class `Monitor`, defined in the Daut library. In this case, the class `P_RSA` extends the class `DautMonitor`, which itself extends class `Monitor`. Class `Monitor` defines, amongst other things, the type `state`, the state producing functions `always` and `watch`, and finally `error` and `ok`, which represent end states with the obvious meanings.

A Daut monitor maintains a set of current states, each of type `state`. Each state is associated with a transition function of type: `PartialFunction[E,Set[state]]`, where `E` is the type of events submitted to the monitor. A partial function can in Scala be defined as a block of **case**-statements, defining the exact domain for which it is defined. A transition function returns a set of new states when applied to an event for which it is defined.[3] Different kinds of states are supported, such as always-states which are always active, and watch-states, which are active until an event matches a transition, at which point they are removed. The functions

[3] Given a partial function f in Scala, the expression $f.isDefined(e)$ is true iff. the function f is defined for the value e, in our case whether event e matches one of the **case**-statements. This is used by the monitor to determine whether a state is enabled to process an event.

```
1  class P_RSA(config: Config) extends DautMonitor(config) {
2    always {
3      case e@Visit(Info(_, track), wp) if isNewFlight(e) ⇒
4        if( isInitialWaypoint (track, wp)) nextState(wp, track.cs) else error()
5      case e@Completed(_) if isNewFlight(e) ⇒ error()
6    }
7
8    def nextState(wp: Waypoint, cs: String): state = {
9      val next = star.next(wp)
10     watch {
11       case Visit(Info(State(`cs`,_,_,_),_),`wp`) ⇒ nextState(wp, cs)
12       case Visit(Info(State(`cs`,_,_,_), _),`next`) ⇒
13         if(next == last) ok else nextState(next, cs)
14       case Visit(Info(State(`cs`,_,_,_),_),_) ⇒ error()
15       case Completed(Track(`cs`,_,_)) ⇒ error()
16       case StarChanged(Track(`cs`,_,_)) ⇒ dropMonitor(cs)
17     }
18   }
19 }
```

Fig. 6 Implementation of Property P_{RSA} in Daut

`always` and `watch` produce such states when applied to partial functions representing the transitions.

The monitor should be read as follows. Lines 2-6 define the initial state, always observing `Visit` and `Completed` events. The partial function in lines 3-5 defines the transitions in this always-state. The first case matches a `Visit` event `e`,[4] with an `Info` argument containing a `track`, and a waypoint `wp`, and where the condition `isNewFlight(e)` is true, meaning that the event represents a new flight (new call sign) not already monitored. In this case, if the waypoint is the first we enter, a new state is entered, returned by a call of the function `nextState`, now monitoring the flight in this initial waypoint. Otherwise (if a not yet monitored flight starts in a waypoint different from the first), an `error` state is entered. If it is a `Completed` event for a new flight, indicating that the flight is completed without visiting any waypoints, an `error` state is entered.

Note that although `nextState`, lines 8-18, is a normal Scala function, introducing this function gives the resemblance of a state machine with two kinds of states, the initial always-state, and the state(s) represented by this function. In Sect. 6 we shall see an example of a property with "unnamed" states, resembling how temporal logic does not refer to named states.

The function `nextState` returns a new state monitoring a specific flight with a specific call signal `cs` and at a certain current waypoint `wp`. First, in line 9, we compute the next waypoint, `next`, reachable from the current waypoint `wp`, used to monitor whether that next waypoint is entered. The subsequent call of the `watch` function, lines 10-17, returns a state monitoring the transitions provided to the call as the partial function in lines 11-16. If a `Visit` event is observed, line 11, with a call sign matching the call sign provided as parameter (grave accent quotes around a variable mean that the value of that variable has to be matched) and with a waypoint matching the current, then we just continue in the current state. If, on the other hand, lines 12-13, it is a `Visit` event where the waypoint matches the `next` way point, then if it is the last we terminate monitoring in the `ok` state, and if not then we continue monitoring in the `next` waypoint. If it is a `Visit` event and none of these cases match, line 14, we end up in the `error` state. If a `Completed` event is observed, line 15, without having reached the end state via a `Visit` event first (line 13), an `error` state is entered. Finally, line 16, if the route is changed for that flight, the flight is dropped as being monitored, and we enter the `ok` state.

4.3 A MESA monitoring system for P_{RSA}

Figure 7 illustrates the MESA monitoring system used to verify Property P_{RSA}. The data acquisition step extracts the data relevant to the property which includes flight information, position, navigation specification, flight plan, etc. To get this data, we connect to an FAA system, SWIM (System Wide Information Management) [27]. SWIM implements a set of information technology principles in NAS which consolidates data from many sources, e.g., flight data, weather data, surveillance data, airport operational status. Its purpose is to provide relevant NAS data, in standard XML formats, to its authorized users such as airlines, and airports. SWIM has a service-oriented architecture, which adopts the *Java Message Service (JMS)* interface [50] as a messaging API to deliver data to JMS clients subscribed to its bus. We use the RACE actor `SFDPS-importer`, which is a JMS client configured to obtain en-route real-time flight data from a SWIM

[4] The pattern `e @` *pattern* behaves as *pattern*, but in addition gives the name `e` to the value matching the *pattern*.

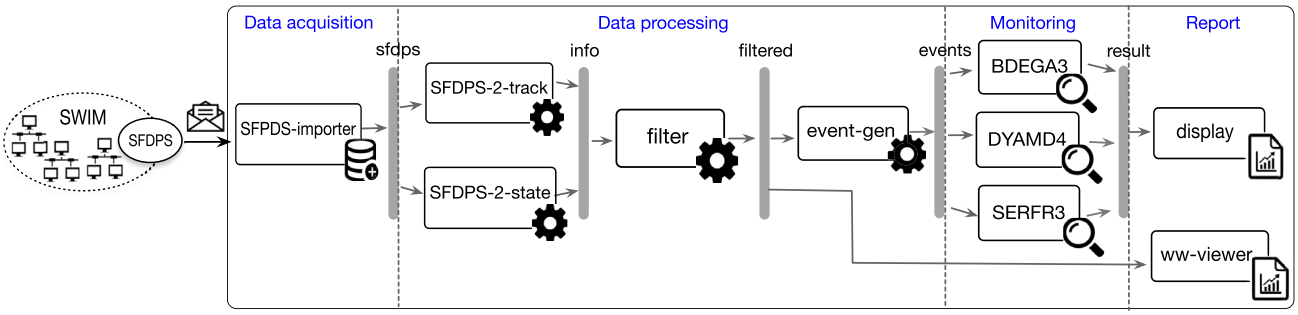


Fig. 7 A MESA instance for verifying Property P_{RSA} for RNAV STARs at SFO

service, SFDPS (SWIM Flight Data Publication Service) [55]. `SFDPS-importer` publishes the data to the channel `sfdps`.

The data processing step parses the SFDPS data obtained from the previous stage by subscribing to the channel `sfdps`, and generates a trace, which is composed of event objects, relevant to the property. This is done via a pipeline of actors that parse the SFDPS messages in XML (`SFDPS-2-track` and `SFDPS-2-state`), filter irrelevant data (`filter`), and finally generate `Visit`, `Completed`, and `StarChanged` events, which are known to the monitor `P_RSA` (`event-gen`) and published to the channel `trace`.

The monitoring step includes monitor actors that encapsulate an instance of the monitor `P_RSA` (Fig. 6). They subscribe to the channel `trace`, and feed their underlying `P_RSA` object with incoming events. Each monitor actor in Fig. 7 is associated to a RNAV STAR procedure at SFO which checks for the flights assigned to that RNAV STAR, and published the verification result on the channel `result`. Using the dispatcher feature of MESA, one can distribute the monitoring differently, for example using one monitor actor per flight. Finally, the last step displays the results. The actor `display` simply prints data published on `result` on the console. We also use a RACE actor, `ww-viewer`, that uses NASA WorldWind system [44] to provide interactive geospatial visualization of flight trajectories.

Using the MESA system shown in Fig. 7, we discovered violations of P_{RSA}. Figure 8 includes snapshots from our visualization illustrating two cases where P_{RSA} was violated. In both cases, the flights are assigned to a route in the BDEGA3 procedure. As shown in Fig. 8, the flight United 1738 missed the waypoint LOZIT, and the flight Jazz Air 743 missed the initial waypoint BGGLO.

Fig. 8 Flight deviation from the assigned RNAV STARs detected at SFO

5 Akka threading model

Since a main focus of this work is evaluating the impact of concurrent monitors, it is essential to understand the Akka threading model. This section explains the underlying threading model in Akka to show how actors in Akka are scheduled. Scheduling actors in Akka is performed by low-level components built into the Akka toolkit, which are referred to as dispatchers. Note that these dispatchers are completely different from the dispatcher components implemented in MESA. To avoid confusion, in some contexts, we refer to the ones implemented by Akka as *Akka dispatchers*. Akka dispatchers are responsible for management of actor mailboxes and the threading strategy. They push messages into actors mailboxes, and associate threads from the thread pools to actors to process messages in their mailboxes. Akka provides a fixed number of dispatchers to choose from. The user can

also assign a certain Akka dispatcher to a group of actors. The built-in dispatcher types in Akka are as follows.

- `Dispatcher` is the default Akka dispatcher which associates all the assigned actors to one thread pool.
- `PinnedDispatcher` provides an actor with exclusive access to a single thread.
- `BalancingDispatcher` redistributes messages from busy actors to the ones with empty mailboxes.
- `CallingThreadDispatcher` is only used for testing, and uses the current thread to execute any actor.

Akka provides configuration parameters to tune dispatchers to specific needs. The parameter `throughput` represents the maximum number of messages processed by the actor before the assigned thread is returned to the pool. The parameter `throughput-deadline-time` represents the deadline for the actor to process messages each time it executes. One can also specify the underlying thread pool implementation used in the Akka dispatcher by setting its executor component using the parameter `executor`. By default, Akka uses `fork-join-executor`, which relies on the *work-stealing* pattern where threads always try to find tasks from the submitted tasks to the pool and the ones created by other running tasks. Akka also includes `thread-pool-executor`, which offers a dynamic thread pool that can decrease or increase in size depending on how busy or idle the threads are. Akka also allows users to implement their own customized executor.

The number of threads in the pool is another measure that can be tuned. With too few threads, which may cause low CPU utilization, the actors are not able to keep up with the arrival of messages. With too many threads, the context switch time between threads increases, which leaves less time for processing the threads. The Akka dispatcher configuration provides three parameters to specify the thread pool size. The parameters `parallelism-min` and `parallelism-max` represent the minimum and maximum number of threads, respectively, and `parallelism-factor` is a factor to calculate the number of threads based on available processors. The size of the thread pool is `parallelism-factor` multiplied by the number of available processors. The number of available processors is the value returned by the method `java.lang.Runtime.availableProcessors()` which gives the maximum number of logical cores available to the virtual machine. If the calculated thread pool size is smaller than `parallelism-min` or larger than `parallelism-max`, then the thread pool size becomes `parallelism-min` or `parallelism-max`, respectively. Moreover, to set the thread pool size to a specific value, one could set both `parallelism-min` and `parallelism-max` to that value.

To find the right configuration, one needs to benchmark with different dispatcher parameters presented in this section. Section 6.2, which presents the setup for our experiments, includes the values used for the Akka dispatcher parameters in our experiments. We use the same configuration for all the experiments.

6 Experiments

This section presents our experiments evaluating the impact of using concurrent monitors and indexing. More details on the experiments can be found in [53]. The experiments use a property which checks if the sequence of SFDPS messages with the same call sign received from SWIM is ordered by the time tag attached to the messages. This property is motivated by observations where the SFDPS messages did not send in the right order by SWIM. The reason for considering this property for our experiments, instead of P_{RSA} (Sect. 4), is that, unlike P_{RSA}, this property applies to all flights, which provides us with a larger data set. It should also be noted that our analysis in Sect. 4 is based on the assumption that messages are always received in the correct order. We use the state of flights as events captured by `State` instances, and specify the property, named P, in Daut as follows, where t1 and t2 represent the event time.

```
class P(config: Config) extends DautMonitor(config){
  always {
    case State(cs,_,_,t1) ⇒ watch {
      case State('cs', _,_,t2) ⇒ t2.isAfter(t1)}
  }
}
```

The monitor reads as follows: It always holds that if a `State` event is observed for a flight with call sign `cs` at a time `t1`, then the next observed `State` event after that, with the same call sign `cs`, must have a time stamp `t2` which is after `t1`, determined by the boolean expression `t2.isAfter(t1)`. A boolean expression in Daut, occurring at a position where a result state is expected, is interpreted as `ok` if true and as `error()` if false. Note that in contrast to the monitor in Fig. 6 the `watch` function is applied immediately after the first => arrow, without defining a new function containing this call (similar to giving a name to the result state). It corresponds to simply inlining the body of the function one could have otherwise written. This is an example of how one can write monitors with a temporal logic flavor in contrast to a state machine flavor as in Fig. 6.

The property P is simple, and it leads to a small *service time*, the time used to process the message within the monitor object. To mitigate issues associated with microbenchmarking, we use a feature of Daut that allows for defining sub-monitors within a monitor object. We implement a Daut monitor `P_SEQ` as follows, which maintains a list of sub-monitors, all monitoring the same property P.

```
class P_SEQ(config: Config) extends DautMonitor(config){
  val size = config.getInt("sub-monitor-count") - 1
  val m = for(i ← 0 to size) yield new P(config)
  monitor(m: _*)
}
```

The variable `size` is assigned to the desired number of sub-monitors, denoted by the key `sub-monitor-count` in the configuration map, which is set by the user. Next, `m` is assigned a list of `size` monitor instances, each monitoring property P, using a **for**-expression (Scala's version of a list comprehension). Finally, the function `monitor` is applied to this list (the `_*` is needed to turn the list `m` into a variable-length argument list[5]). The `monitor` function adds each of its argument monitors as a sub-monitor.

We evaluate the impact of concurrency in the context of indexing. Indexing is an optimization technique that can be applied both at the monitor level or the dispatcher level, and serves to reduce the number of states searched when a new event is submitted. Indexing at the monitor level is supported by Daut. We activate this feature by overriding the indexing function `keyOf` in the Daut monitor. This function, when applied to an event, returns a key to index on, in this case the call sign (flight identifier) for each event.

```
override protected def keyOf(event: Any) = {
  e match {
    case State(cs, _, _, _) ⇒ Some(cs)
  }
}
```

The Daut monitor will subsequently internally organize the states in a hash map, mapping each key to the states relevant for events with that key. Given an observed event, the monitor obtains the key of the event by applying `keyOf`, and looks up the states mapped to by that key, which are then applied to the event, instead of iterating over all the current states.

At the dispatcher level, indexing is applied by keeping the monitor instances or references to monitor actors in a hash map, using the call signs carried by events as entries to the hash map.

6.1 Monitoring systems

The experiments use four different MESA systems, which are illustrated in Fig. 9 (the last option represents two different monitoring systems, as explained below). Each system monitors exactly one property. It can be seen that all the systems have the same data acquisition and data processing phases, and they are only different in their monitoring phase. They use the `instant-reply` actor to acquire the input data, and the `event-gen` actor to process the data and create the trace for the monitors. The `instant-reply` actor accesses an archive containing recorded SFDPS data messages in the XML format, and as it reads the messages, it publishes them to the `sfdps` channel instantly. The `event-gen` actor obtains the XML messages by subscribing to the channel `sfdps`, generates a trace composed of `State` objects from the SFDPS data, and publishes the `State` objects to the channel `events` accessed in the monitoring step.

Let `n` be the total number of different call signs in the input sequence. The outermost white boxes represent actors, and gray boxes represent monitor instances held by the actor. Let `M` refer to `P_SEQ` monitor instances with no indexing capability, and `MI` refer to `P_SEQ` instances with indexing. The white box inside each monitor instance includes call signs monitored by this instance. Next, we explain the monitoring step for the monitoring systems.

- `monitor-indexing` - the monitoring step includes one actor with a single `MI` monitor which checks for all the events in the input sequence published to `events`. The monitoring step of this configuration is equivalent to directly using the Daut tool to process the trace sequentially, with indexing occurring in Daut.
- `dispatcher-indexing` - the monitoring step includes a dispatcher actor which creates monitor instances of type `M`, and feeds them with incoming events. The dispatcher actor generates one monitor instance per call sign, and applies indexing by storing the monitor instances in a hash map. The dispatcher obtains event objects from the channel `events`, and, starting with an empty hash map, for each new call sign, it adds a new monitor instance to the hash map. For an event object with the call sign $\texttt{cs}_\texttt{i}$, the dispatcher invokes the `verify` method of the monitor instance $\texttt{M}_\texttt{i}$.
- `concurrent` - the trace analysis is performed concurrently by employing multiple monitor actors, generated on-the-fly. This demonstrates how a single property is checked by multiple concurrent monitors. One can configure the dispatcher to set a limit on the number of monitor actors. If no limit is set, one monitor actor is generated for each call sign and the indexing within the monitor is deactivated. This monitoring system is referred to as `unbounded-concurrent`. By setting a limit, one monitor actor could be assigned to more than one call sign. Such monitoring system is referred to as `bounded-concurrent`. Indexing is also applied at the dispatcher level, using a hash map that stores monitor actor references with call signs as entries to the map. For each event object, the dispatcher forwards the event object to the associated monitor actor via point-to-point communication. Then the monitor actor invokes the `verify` method on its underlying monitor instance.

The main features of the monitoring systems are summarized in Fig. 10. The rows represent the monitoring systems. The first and second columns show if indexing is

[5] A variable-length argument list refers to a list of arguments of arbitrary length, all of the same type.

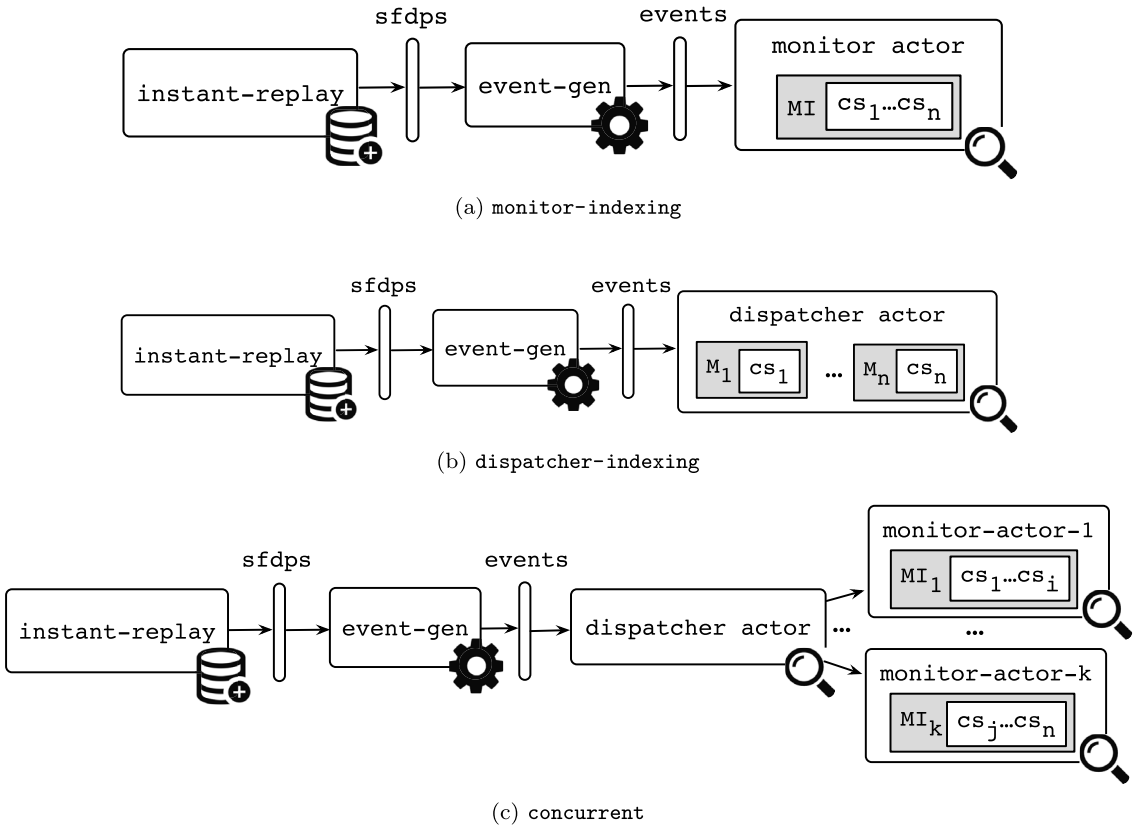


Fig. 9 Actor-based monitoring systems used in the experiment

Fig. 10 The main features of the monitoring systems presented in Fig. 9

	monitor indx	dispatcher indx	concurrency
monitor-indexing	✓	×	×
dispatcher-indexing	×	✓	×
unbounded-concurrent	×	✓	✓
bounded-concurrent	✓	✓	✓

applied at the monitor level and the dispatcher level, respectively. The third column shows if the monitoring system includes concurrent monitor actors. The monitoring systems `dispatcher-indexing` and `monitor-indexing` are similar except for their indexing mechanisms. In both systems, the monitoring phase includes only one actor that performs the monitoring task sequentially.

6.2 System setup

All experiments were performed on an Ubuntu 18.04.3 LTS machine, 31.1 GB of RAM, using a Intel®Xeon®W-2155 CPU (10 cores with hyperthreading, 3.30 GHz base frequency). We use an input trace, `T`, including 200,000 messages obtained from an archive of recorded SFDPS data in all experiments. `T` includes data from 3215 different flights, that is, `n` in Fig. 9 is 3215. The number of sub-monitors in `P_SEQ` is set to 2000. The Java heap size is set to 12 GB. We also use the default Akka dispatcher setting in the experiment, which is as follows. All actors use the `Dispatcher` implementation with the default value 5 for `throughput`, 0 for `throughput-deadline-time`, which indicates no time limit, and `fork-join-executor` for the executor. Moreover, `parallelism-min` and `parallelism-max` are set to 8 and 64, and `parallelism-factor` is set to 3 which leads to the thread pool of size 30 (`parallelism-factor` × number of cores) in the machine with 10 cores. It should be noted that changing the configuration can impact the results. One can benchmark to find the right configuration for each setting. However, since our evaluation involves systems with different numbers of concurrent components, to simplify, we use one configuration for all the experiments.

6.3 Evaluation

Since garbage collection in the Java Virtual Machine (JVM) is beyond our control, each experiment needs to be repeated

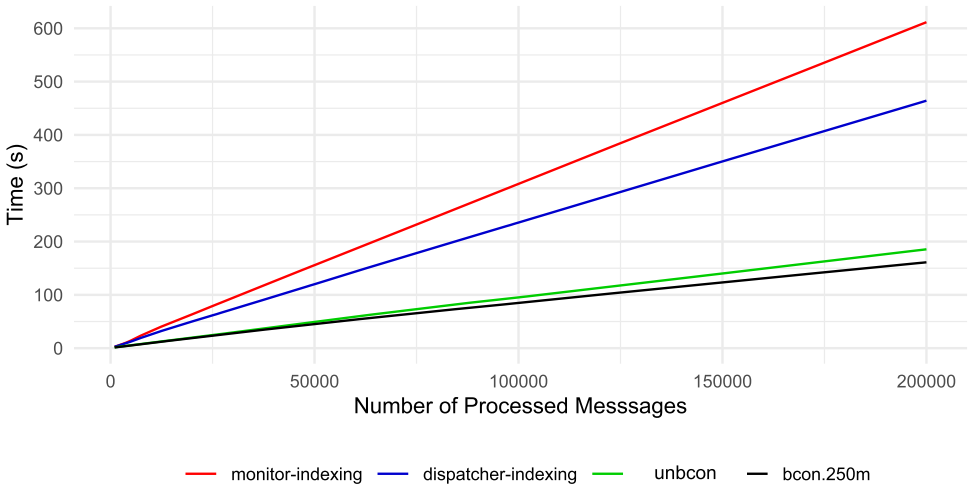
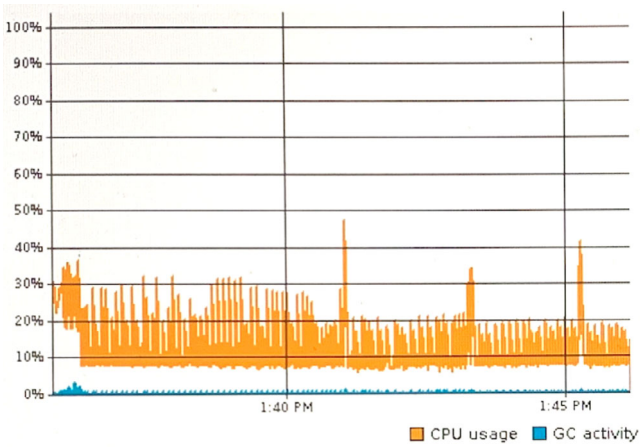
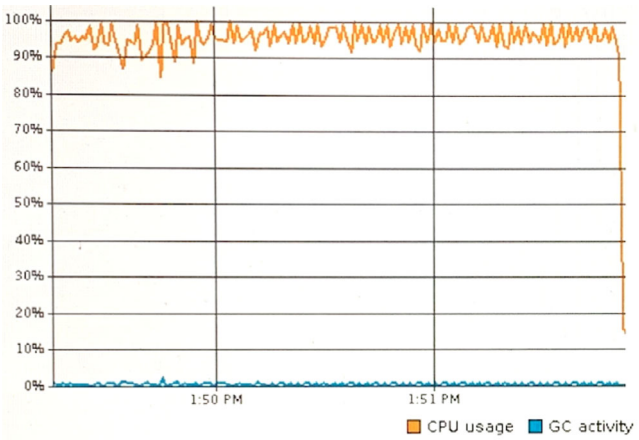


Fig. 11 Comparing the run times of different MESA actor systems



(a) `monitor-indexing`



(b) `bounded-concurrent`(250 monitors)

Fig. 12 The CPU utilization profiles obtained by VisualVM

several times. Using a bash script, each MESA monitoring system is run 10 consecutive times on the trace `T`, and the average of the runs is used for evaluation. Figure 11 compares the run times for the monitoring systems presented in Fig. 9. The legend `unbcon` stands for `unbounded-concurrent`, and the legend `bcon` stands for `bounded-concurrent`, followed by the number of monitor actors. Considering the 3215 different call signs in `T`, `monitor-indexing` includes one monitor actor including one monitor object that tracks all 3215 flights. The `dispatcher-indexing` system creates one actor with a hash map of size 3215 storing the monitor objects, where each object monitors events from one flight. The `unbounded-concurrent` monitoring system creates 3215 monitor actors where each actor monitors events from one flight. The `bounded-concurrent` system creates 250 monitor actors, where each actor monitors events from 12 or 13 flights.

The results show that the systems with concurrent monitors perform considerably better than the systems with a single monitor actor. The system `monitor-indexing` performs worse than `dispatcher-indexing`. The difference amounts to a larger indexing overhead in `monitor-indexing`. Since the number of sub-monitors captured by `MI` is 2000, indexing at the monitor level is repeated 2000 times per incoming event. This leads to a higher overhead comparing to indexing at the dispatcher level in `dispatcher-indexing` which is only performed once. The CPU utilization profiles for the system are obtained by the VisualVM profiler, which represent the percentage of total computing resources in use during the run (Fig. 12). The CPU utilization for `monitor-indexing` is mostly under 30% and for `dispatcher-indexing` is mostly between 40% and 50%. For `unbounded-concurrent` and `bounded-concurrent`, the CPU utilization is mostly above 90% which shows the impact of using concurrent monitor actors. The VisualVM heap data profiles reveal that all the

Fig. 13 Comparing the run times of different MESA actor systems

#monitors	125m	250m	500m	1000m	2000m	3215m
time (s)	169	161	167	169	183	208

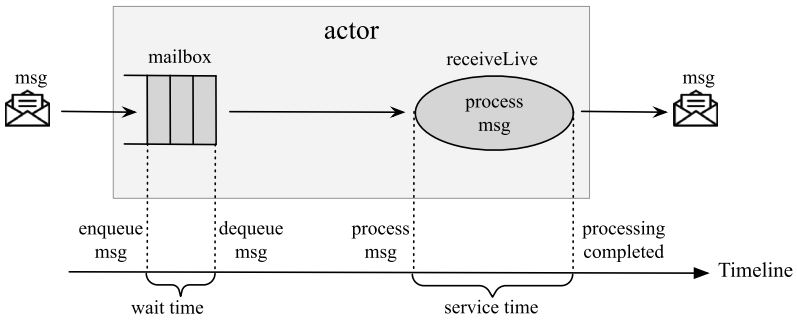


Fig. 14 The timeline for a message sent to the actor [51]

system exhibit a similar heap usage, which mostly remains under 10G.

Figure 11 shows that limiting the concurrent monitors to 250 results in a better performance than using one monitor actor per flight in `unbounded-concurrent`. To evaluate how the number of monitor actors impact the performance, `bounded-concurrent` is run with different numbers of monitor actors, 125, 250, 500, 1000, 2000, and 3215. We increase the number of monitor actors up to 3215, since this is the number of total flights in the trace `T`. The results are compared in Fig. 13. The system performs best with 250 monitor actors, and from there as the number of monitor actors increases, the run time increases. Increasing the number of monitor actors decreases the load on each monitor actor, however, it increases the overhead from their scheduling and maintenance. Note that the optimal number of monitor actors depends on the application and the value of input parameters. Tweaking inputs parameters could lead to a different optimal number of monitor actors. Our results also show that depending on the number of flights tracked by each monitor actor, Daut indexing can lead to overhead. For e.g., it leads to 11% overhead (compared to not using Daut indexing) when using 3215 monitor actors (since indexing is performed, but it is not needed). On the other hand, Daut indexing leads to performance improvement by 45% (compared to not using Daut indexing) when using 125 monitor actors (since indexing is beneficial in this case).

6.4 Actor parameter evaluation

To investigate the underlying factors behind runtime results further, we also evaluate performance parameters for individual dispatcher and monitor actors. The performance parameters that we consider include the average service time, the average wait time for messages in the actor's mailbox, and the average size of the actor's mailbox queue obtained after a message is enqueued. Note that this does not exactly reflect the average size of the mailbox, since it does not take into account the mailbox changes in between enqueues, where mailbox queue can stay empty for a while.

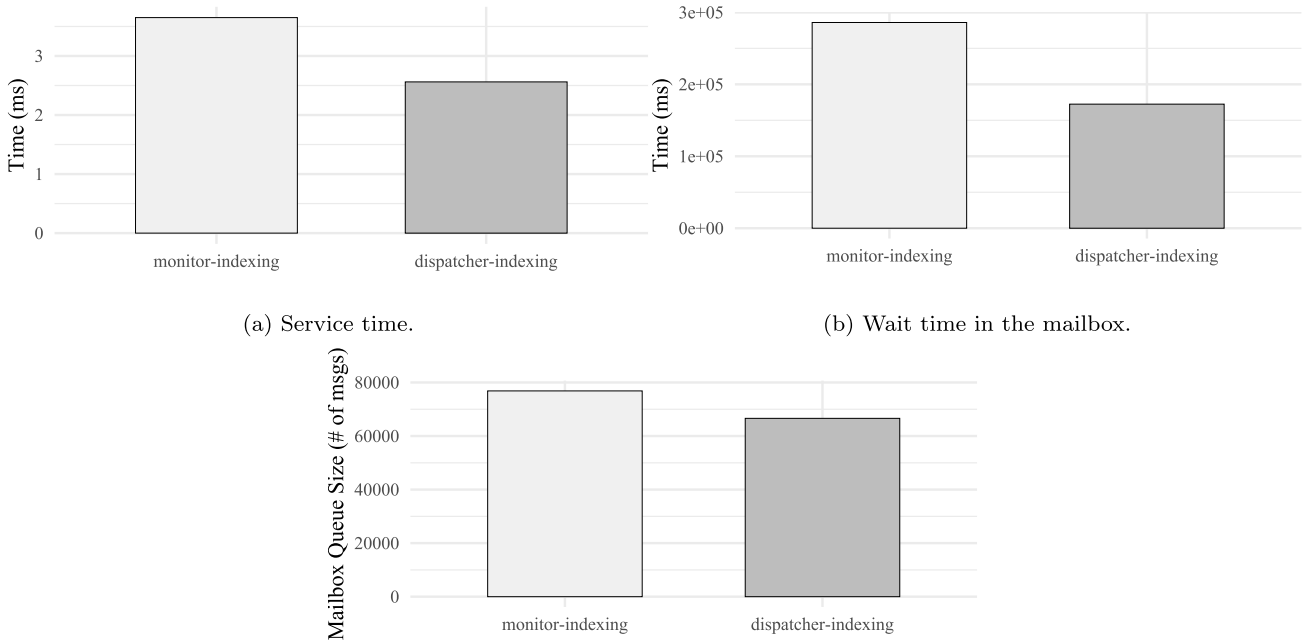
Figure 14 illustrates the relevant points at which we record data to measure the actors performance metrics. The method `recieveLive`, which processes the incoming message, captures the default actor behavior. The interesting points for measuring these parameters are when a message is enqueued into and dequeued from the mailbox, and when the actor starts processing and finishes processing a message. We provide mechanisms for actors to wrap the relevant data into container objects, which are defined as `case` classes, and publish them to a channel accessed by an actor, `stat-collector`, which collects this information and reports when the system terminates.

To measure service time, the default actor behavior, `recieveLive`, is replaced by an implementation that for each message, invokes `recieveLive`, records the time before and after the `recieveLive` invocation, and publishes a data container with the recorded times to the channel accessed by the `stat-collector` actor. The approach that we used to collect mailbox data is similar to the one proposed in Chap. 16 of [51] which discusses how one can customize and configure Akka to improve the overall performance. To obtain information from actor mailboxes, we implement a new mailbox type, called `StatsMailboxType`, that extends the default Akka mailbox implementation with a mechanism that records the message entry time to and the exit time from the mailbox, and the size of the mailbox queue after a message is enqueued, and publishes a data container with the recorded data to the channel accessed by the `stat-collector` actor.

Any MESA actor can be configured to use these features, referred to as ASF, which stands for Actor Statistics Features. A MESA actor has the `boolean` property `stats.service` which is false by default. To activate the feature that collects the service time, one needs to set `stats.service` to `true` in the actor configuration. To activate the feature that collects mailbox data, the new mailbox type is specified by adding the following lines in the configuration file.

system	monitor-indexing	dispatcher-indexing	125m	250m	500m	1000m	2000m	3215m
overhead	20%	11%	20%	22%	21%	27%	29%	28%

Fig. 15 Overhead from activating the ASF features



(a) Service time.

(b) Wait time in the mailbox.

(c) Size of the mailbox queue.

Fig. 16 Comparing the actors performance metrics for the `monitor-indexing` and `dispatcher-indexing` systems

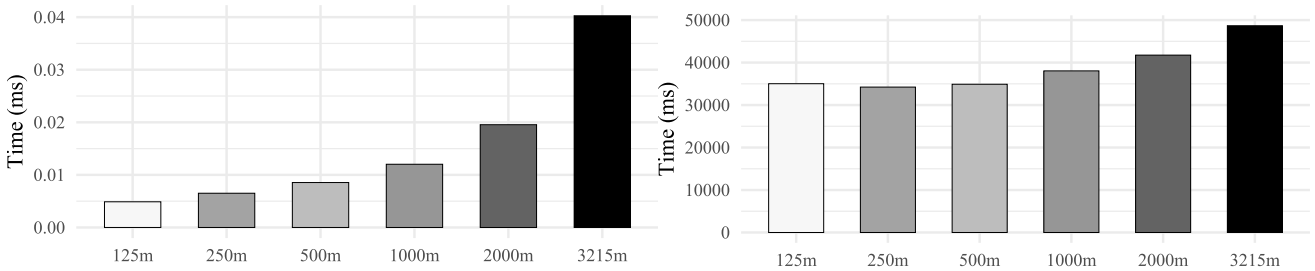
```
stats−collector−mailbox {
  mailbox−type =
        "gov.nasa.mesa.reporting.stats.StatsMailboxType"
}
```

The experiments presented in this section also use the setup outlined in Sect. 6.2. The table in Fig. 15 presents the run time overhead from activating ASF. It can be seen that the ASF overheads for `monitor-indexing` and `dispatcher-indexing` are about 20% and 11%. For systems with concurrent monitor actors, this overhead ranges between 20% to 28% and, overall, increases as the number of monitor actors increases.

Figure 16 compares the performance parameters for individual actors for the `monitor-indexing` and `dispatcher-indexing` systems. Figure 16a and 16b show that the monitor actor in `monitor-indexing` has a longer service time, and a longer wait time in the mailbox comparing to the dispatcher in `dispatcher-indexing`. Moreover, Fig. 16c shows more messages accumulate in the monitor-actor mailbox in the `monitor-indexing` system comparing to the dispatcher mailbox in the `dispatcher-indexing` system. These observations are aligned with the fact that indexing in `monitor-indexing` introduces a higher overhead since for each incoming event, it is performed once per sub-monitor, that is, 2000 times in this example, whereas indexing at the dispatcher level in `dispatcher-indexing` is only performed once per incoming event.

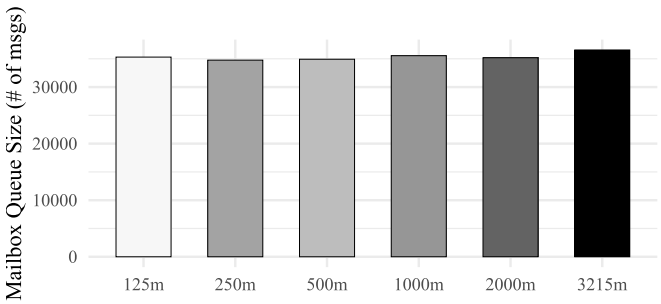
Figure 17 compares the dispatcher actors performance metrics for `bounded-concurrent` with different numbers of monitor actors. It can be seen from Fig. 17a that the average service time increases as the number of actors increases. This is aligned with the fact that using more monitor actors increases the load of the dispatcher actor since it needs to generate more monitor actors. To mitigate this effect, one could introduce multiple dispatchers, which is not evaluated in our study. Moreover, Fig. 17b shows that starting from the system with 500 monitors, the average message wait time in the queue increases as the number of actors increases. In general, with a constant thread pool size, increasing the number of actors in the system can increase the wait for actors to get scheduled, leading to longer wait for messages in mailboxes. Figure 17c shows that the average mailbox size at the enqueue time for the dispatcher actors in all cases only varies in a very small range.

Figure 18 compares the monitor actors performance metrics for `bounded-concurrent` with different numbers of



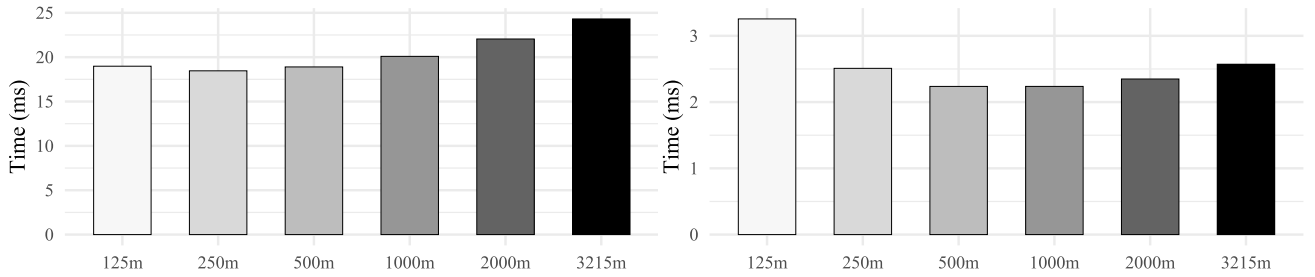
(a) Service time in the dispatcher.

(b) Wait time in the dispatcher mailbox.



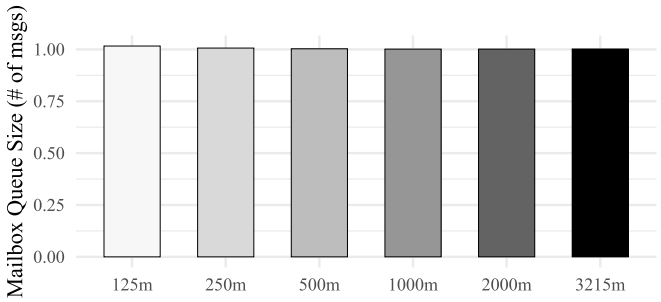
(c) Size of the dispatcher mailbox queue.

Fig. 17 Comparing the dispatcher actors performance metrics for the `bounded-concurrent` systems



(a) Service time in the monitor actor.

(b) Wait time in the monitor actor mailbox.



(c) Size of the monitor mailbox queue.

Fig. 18 Comparing the monitor actors performance metrics for the `bounded-concurrent` systems

monitor actors. It can be seen from Fig. 18a that starting from the system with 250 monitor actors, the average service time for monitor actors increases as the number of monitor actors increases. Decreasing the number of monitor actors increases the load on individual actors, since each monitor actor deals with higher number of flights. On the other hand,

applying indexing within the monitor actors helps with improving their performance, however, for monitors that track small number of flights, indexing can lead to overhead leading to longer service times. Figure 18b shows that in the case of monitor actors, the mailbox wait is longer with smaller number of actors, unlike the dispatchers (Fig. 17b). This is due to higher arrival rate of messages in these systems, since each monitor actor is assigned to higher number of flights. It can be also seen from Fig. 18c that the average mailbox size at the enqueue time for monitor actors in all the systems is almost 1. That is, on average, every time a new message is placed in the monitor actor mailbox, there are no other messages in the queue in all cases. Note that, as mentioned before, this measure does not reflect the arrival rate of messages, since it does not take into account the periods where the mailboxes are empty.

7 Discussion

MESA is designed as a generic tool for monitoring event streams using actors and indexing (slicing). It is generic by not providing a monitoring DSL itself, but allowing any such as a plugin. In this work, we have used Daut for our experiments. The tool is applicable for concurrent, indexed monitoring, and can be used as such or it can be used for experiments with concurrent monitoring, as presented in this paper.

We have specifically shown the positive impact of using concurrent monitors combined with indexing for runtime verification. The main question is: is there an advantage in using concurrency in monitoring different slices for a single property. The problem is particularly relevant for monitoring of first-order temporal properties, which require fast lookup of relevant parts of a monitor for each data-carrying event. The answer to this question was not obvious up front. In particular, one study [47] had not been able to clearly show an advantage. The concern was that it would just be too fine-grained an application of concurrency, which would have no advantage due to overhead from thread scheduling.

The positive observation, however, is that it is beneficial to split monitoring of a single property into multiple actors, each processing a subset of the indexes (call signs in this experiment), and within each of these actors again index (on the call signs) a second time. However, we showed that it was inefficient to create an actor for each call sign. Rather, it was found optimal to group the call signs, and create a smaller number of actors, in this case 250 (compared to 3215, the total number of call signs), each processing a subset of the flights (12-13 in our case), also referred to as `bounded-concurrent` in Fig. 9. This shall be compared to the sequential approach to indexing (`monitor-indexing` in Fig. 9), corresponding to using a single actor (no concurrency effectively).

As observed, to maximize the performance, one needs to limit the number of concurrent monitor actors. Due to a variety of overhead sources, the optimal number of actors is application-specific and is challenging to determine a priori. The following factors need to be taken into consideration when configuring values of the related parameters. Limiting the number of monitor actors on a multicore machine can lead to a low CPU utilization. One can elevate the CPU utilization by increasing concurrency. However, there is overhead associated with actors. Assigning actors to threads from the thread pool and context switching between them impose overhead. The combination of such competing factors can make one setting perform better than others.

MESA is a highly configurable platform, and it provides mechanisms for evaluating performance parameters for individual dispatcher and monitor actors. Those can facilitate finding the optimal number of monitor actors to maximize the performance. One can easily tune relevant parameters in the configuration file to evaluate the monitoring systems. While the framework provides features for observing performance, parameter tuning, however, is performed manually. Future work can include automated tuning, either prior to monitoring actual data in the field, or dynamically during monitoring in the field, adjusting to the current situation.

As shown in Fig. 2, our framework runs on top of the Java Virtual Machine (JVM) and relies on the Akka framework. There are mechanisms, such as garbage collection at the JVM level and actor scheduling at the Akka level, that cannot be controlled from a MESA system. Therefore, MESA is not suitable for verifying hard real-time systems where there are time constraints on the system response. One of the challenges that we faced in this work is micro-benchmarking on the JVM, which is a well-known problem. Certain characteristics of the JVM, such as code optimization can impact accuracy of the results, specially when it comes to smaller time measures such as service time and wait time for messages in the actor mailboxes. However, there are tools such as JMH that provide accurate benchmarking [35].

An important issue concerns correctness of the approach. There are two aspects of this issue, namely the correctness of an individual monitor, and the correctness of the approach to concurrency. Wrt. correctness of a monitor, it is common (in contrast to our approach) to automatically synthesize a such from a specification in an external DSL. In this case, the verification problem becomes that of ensuring that the synthesized monitor correctly implements the specification. This problem can be approached top down, synthesizing a correct-by-construction monitor, see e.g., [1, 14], or bottom up, proving a synthesized monitor correct wrt. the speci-

fication, see e.g. [11, 22]. However, in our case, there is no specification in an external DSL from which a monitor is synthesized. The monitor is programmed directly in the Daut library, an internal DSL, developed specifically for writing monitors. This library hopefully makes writing such monitors as easy as would an external DSL offering the same features. The correctness problem in this case consists of proving that the library correctly implements the intended behavior of the internal DSL. Such an effort remains as future work. Finally, validation consists of ensuring that the monitor meets the intention of the user. It is our view that it is fairly easy to convince oneself that the monitor in Fig. 6 and the monitors used for the experiment in Sect. 6 express the desired properties. However, validation will always be an important task. Daut offers debugging features, allowing printing of internal monitor actions and states, which can help in validating a monitor.

Wrt. to correctness of the concurrency approach, first note that the approach is thread-safe in the sense that different actors do not interact in any way beyond all writing to standard output, which does not influence the correctness (although output from different actors may be merged). Second, one can easily convince oneself that all events with the same call sign always end up in the same actor, and that within each actor, events with the same call sign are indexed to the same slice (bucket in the hash map used for indexing). This is due to the fact that distribution on actors and indexing within actors is based on the same key, namely the call sign. This ensures that the properties being checked are correct in the presence of concurrency.

Note, however, that slicing does put a restriction on what properties can be monitored. Since the trace is sliced into subtraces, each of which may be submitted to its own actor, one cannot express properties that relate different slices. An example of a property that cannot be stated in, e.g., this particular case study is that the route taken by an airplane depends on the routes taken by other airplanes. In MESA, the slicing strategy is manually defined, and attention must be paid to the property being verified to ensure a sound approach.

Even though the focus of the case study has been air traffic routes, and use of a specific monitor, we believe that the main result, that it is beneficial to monitor slices of a single property in multiple actors, transfers to the general problem of monitoring events that carry data. It is of course hard to generalize such a result conclusively, but the result is convincing enough that it may trigger other studies for other cases. For e.g., one case to explore, and which we have not covered, is that of monitoring many different properties, each associated with a dispatch actor (see Fig. 9), under which multiple sub-actors handle the different slices of a single property.

8 Conclusion

We have presented MESA, a runtime verification framework for indexed concurrent monitoring with actors. MESA can be instantiated with different monitoring logics. In this paper, we specified properties in the Daut monitoring logic (library) supporting a mix of data-parameterized state machines and a form of temporal logic. We illustrated MESA by presenting a case study, which obtains live air traffic data feeds and verifies that flights adhere to assigned arrival routes. We then performed an empirical study to evaluate different combinations of concurrency and indexing applied at different levels. We observe, as the main result, that there are clear benefits to monitor a single property with multiple concurrent actors processing different slices of the input trace. This is not an obvious result, since there is a cost to scheduling of small tasks.

Future work includes confirming the result on other examples. Furthermore, it might be worth experimenting with dynamic automated optimization of parameters that determine how many actors to create for a single property. Finally, experiments need to be done for the case where multiple different properties are monitored, each evaluated with indexing.

Acknowledgements The research performed by the second author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An operational guide to monitorability with applications to regular properties. Softw. Syst. Model. **20**(2), 335–361 (2021). https://doi.org/10.1007/s10270-020-00860-z
2. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: On benchmarking for concurrent runtime verification. In: Guerra, E., Stoelinga, M. (eds.) Fundamental Approaches to Software Engineering, pp. 3–23. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-71500-7_1
3. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pp. 239–252 (2016). https://doi.org/10.1109/CSF.2016.24
4. Akka (2020). http://doc.akka.io/docs/akka/current/scala.html
5. Artho, C., Havelund, K., Kumar, R., Yamagata, Y.: Domain-specific languages with Scala. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 9407, pp. 1–16. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-25423-4_1
6. Attard, D.P., Francalanza, A.: Trace partitioning and local monitoring for asynchronous components. In: Cimatti, A., Sirjani, M. (eds.) International Conference on Software Engineering and Formal Methods. Lecture Notes in Computer Science, vol. 10469, pp. 219–235. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-66197-1_14

7. Avrekh, I., Matthews, B.L., Stewart, M.: RNAV adherence data integration system using aviation and environmental sources. Tech. rep., NASA Ames Research Center (2018)
8. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: Qadeer, S., Tasiran, S. (eds.) International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 7687, pp. 184–198. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-35632-2_20
9. Barringer, H., Havelund, K.: TraceContract: a Scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 6664, pp. 57–72. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21437-0_7
10. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specification. Form. Methods Syst. Des. **49**, 75–108 (2016). https://doi.org/10.1007/s10703-016-0242-y
11. Basin, D.A., Dardinier, T., Heimes, L., Krstic, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. Lecture Notes in Computer Science, vol. 12166, pp. 432–453. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-51074-9_25
12. Basin, D., Gras, M., Krstić, S., Schneider, J.: Scalable online monitoring of distributed systems. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020. Lecture Notes in Computer Science vol. 12399, pp. 197–220. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-60508-7_11
13. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime verification with minimal intrusion through parallelism. Form. Methods Syst. Des. **46**, 317–348 (2015). https://doi.org/10.1007/s10703-015-0226-3
14. Burlò, C.B., Francalanza, A., Scalas, A.: On the monitorability of session types, in theory and practice (extended version). CoRR (2021). arXiv:2105.06291. https://doi.org/10.4230/LIPIcs.ECOOP.2021.20
15. Clarkson, M.R., Schneider, F.B.: Hyperprop. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
16. 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.) Software Engineering and Formal Methods. Lecture Notes in Computer Science, vol. 7504, pp. 218–232. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-33826-7_15
17. Department of Transportation, Federal Aviation Administration: Implementation of Descend via into Boston Terminal area from Boston ARTCC (2015)
18. El-Hokayem, A., Falcone, Y.: Can we monitor all multithreaded programs? In: Colombo, C., Leucker, M. (eds.) International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 11237, pp. 64–89. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_6
19. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. **29**(1), 1:1–1:57 (2020). https://doi.org/10.1145/3355181
20. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press, Amsterdam (2013). https://doi.org/10.3233/978-1-61499-207-3-141
21. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Form. Methods Syst. Des. **54**(3), 336–363 (2019). https://doi.org/10.1007/s10703-019-00334-z
22. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified Rust monitors for Lola specifications. CoRR (2020). arXiv:2012.08961. https://doi.org/10.1007/978-3-030-60508-7_24
23. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Form. Methods Syst. Des. **46**(3), 226–261 (2015). https://doi.org/10.1007/s10703-014-0217-9
24. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
25. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: Reger, G., Havelund, K. (eds.) An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, Seattle, WA, USA, September 15, 2017. Kalpa Publications in Computing, vol. 3, pp. 81–88. EasyChair (2017). https://doi.org/10.29007/4cth
26. Hallé, S., Khoury, R., Gaboury, R.: Event stream processing with multiple threads. In: Lahiri, S., Reger, G. (eds.) International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 10548, pp. 359–369. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_22
27. Harris Corporation: FAA Telecommunications Infrastructure NEMS User Guide (2013)
28. Havelund, K.: Data automata in Scala. In: Symposium on Theoretical Aspects of Software Engineering Conference, Changsha, China, pp. 1–9 (2014). https://doi.org/10.1109/TASE.2014.37
29. Havelund, K.: Daut (2022). https://github.com/havelund/daut
30. Havelund, K.: TraceContract (2022). https://github.com/havelund/tracecontract
31. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 235–245. Kaufmann, San Francisco (1973)
32. HOCON: Human Optimized Config Object Notation (2020). https://github.com/typesafehub/config/blob/master/HOCON.md
33. International Air Line Pilots Associations: FAA Suspends OPD Arrivals for Atlanta International Airport (2016)
34. International Civil Aviation Organization (ICAO): Performance-based Navigation (PBN) Manual, 3rd edn. (2008)
35. JMH - Java Microbenchmark Harness (2020). https://openjdk.java.net/projects/code-tools/jmh/
36. Joyce, J., Lomow, G., Slind, K., Unger, B.: Monitoring distributed systems. ACM Trans. Comput. Syst. **5**(2), 121–150 (1987). https://doi.org/10.1145/13677.22723
37. Kurklu, E., Havelund, K.: A flight rule checker for the LADEE Lunar spacecraft. In: Pun, V.K.I., Stolz, V., Simao, A. (eds.) Theoretical Aspects of Computing - ICTAC 2020, pp. 3–20. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-64276-1_1
38. Lavery, P., Watanabe, T.: An actor-based runtime monitoring system for web and desktop applications. In: Hochin, T., Hirata, H., Nomiya, H. (eds.) International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 385–390. IEEE Comput. Soc., Los Alamitos (2017). https://doi.org/10.1109/SNPD.2017.8022750
39. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
40. Mehlitz, P.: RACE (2022). http://nasarace.github.io/race/
41. Mehlitz, P., Shafiei, N., Tkachuk, O., Davies, M.: RACE: building airspace simulations faster and better with actors. In: Digital Avionics Systems Conference (DASC), pp. 1–9 (2016). https://doi.org/10.1109/DASC.2016.7777991
42. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. **14**(3), 249–289 (2012). https://doi.org/10.1007/s10009-011-0198-6
43. MESA - MEssage-based System Analysis (2022). https://github.com/NASA-SW-VnV/mesa

44. NASA WorldWind (2022). https://worldwind.arc.nasa.gov/
45. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: Wu, P., Hack, S. (eds.) International Conference on Compiler Construction, pp. 98–108. ACM, New York (2017). https://doi.org/10.1145/3033019.3033031
46. Rasmussen, S., Kingston, D., Humphrey, L.: A brief introduction to unmanned systems autonomy services (UxAS). In: 2018 International Conference on Unmanned Aircraft Systems (ICUAS), pp. 257–268 (2018). https://doi.org/10.1109/ICUAS.2018.8453287
47. Reger, G.: Rule-based runtime verification in a multicore system setting. Master's thesis, University of Manchester (2010)
48. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9035, pp. 596–610. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46681-0_55
49. Reger, G., Rydeheard, D., Barringer, H.: MAIL - an interaction layer for exploring the use of multicore in runtime monitoring. (unpublished)
50. Richards, M., Monson-Haefel, R., Chappell, D.A.: Java Message Service, 2nd edn. O'Reilly Media, Inc., Newton (2009)
51. Roestenburg, R., Bakker, R., Williams, R.: Akka in Action, 1st edn. Manning Publications Co., Greenwich (2015)
52. Shafiei, N., Havelund, K., Mehlitz, P.: Actor-based runtime verification with MESA. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification, pp. 221–240. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_12
53. Shafiei, N., Havelund, K., Mehlitz, P.: Empirical Study of Actor-based Runtime Verification. Tech. rep., NASA Ames Research Center (2020)
54. Stewart, M., Matthews, B.: Objective assessment method for RNAV STAR adherence. In: DASC: Digital Avionics Systems Conference (2017). https://doi.org/10.1109/DASC.2017.8102034
55. SWIM flight data publication service (2020). https://www.faa.gov/air_traffic/technology/swim/sfdps/
56. U.S. Department of Transportation. Federal Aviation Administration: Performance Based Navigation PBN NAS Navigation Strategy (2016)
57. U.S. Department of Transportation. Federal Aviation Administration: Instrument Procedures Handbook (IPH) (2017)