Actor-based Runtime Verification with MESA

Nastaran Shafiei[1], Klaus Havelund[2⋆], and Peter Mehlitz[1]

[1] NASA Ames Research Center/SGT Inc., Moffett Field, CA 94035
{nastaran.shafiei,peter.c.mehlitz}@nasa.gov
[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109
klaus.havelund@jpl.nasa.gov

Abstract. This work presents a runtime verification approach implemented in the tool MESA (MEssage-based System Analysis) which allows for using concurrent monitors to check for properties specified in data parameterized temporal logic and state machines. The tool is implemented as an internal Scala DSL. We employ the actor programming model to implement MESA where monitors are captured by concurrent actors that communicate via messaging. The paper presents a case study in which MESA is used to effectively monitor a large number of flights from live US airspace data streams. We also perform an empirical study by conducting experiments using monitoring systems with different numbers of concurrent monitors and different layers of indexing on the data contained in events. The paper describes the experiments, evaluates the results, and discusses challenges faced during the study. The evaluation shows the value of combining concurrency with indexing to handle data rich events.

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 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 [27,16]. There has been a wide variety of work focusing on verifying distributed systems including dynamic verification techniques such as runtime verification [29,14] 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 formulas, finite state machines, regular expressions, etc. Some of the proposed runtime verification techniques related to distributed computing themselves employ a distributed system for monitoring for a variety of reasons such as improving efficiency [8,18,5,9,17,11]. Exploiting parallelism, one can use additional hardware resources for running monitors to reduce their online overhead

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

[9]. Moreover, using concurrent monitors instead of one monolithic monitor, one can achieve higher utilization of available cores [17].

In this paper, we propose a runtime verification approach for analyzing distributed systems which itself is distributed. Our approach is generic and is not tied to a particular SUO. It is motivated by a use case which aims to analyze flight behaviors in the *National Airspace System* (NAS). 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 19000 airports including public, private, and military airports, and up to 5000 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, 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 volume 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*, a model for building concurrent systems. The actor model was proposed in 1973 as a way to deal with concurrency in high performance systems [23]. Concurrent programming is notoriously difficult due to concurrency errors such as race conditions and deadlocks. These errors occur due to lack of data encapsulation to avoid accessing objects' internal state from outside. Thus, mechanisms are required to protect objects' state such as blocking synchronization constructs which can impact scalability and performance. The actor programming model offers an alternative which eliminates these pitfalls. Primary building blocks in the actor programming model are *actors*, which are concurrent objects that do not share state 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 *mailbox*. The model also guarantees that each runnable actor only executes in one thread at a time, a property which allows to view an actor's code as a sequential program.

We create a framework, MESA, using the Akka toolkit [1,38], which 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 facilitates the implementation of actor-based systems that refrain users from dealing with the complexity of key mechanisms such as scheduling actors and communication. We also use the Runtime for Airspace Concept Evaluation (RACE) [31,30] framework, another system built on top 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 [6,22] and Daut (Data automata) [20,21], implemented as domain specific languages (DSLs) [2]. TraceContract, which was also used for command sequence verification in NASA's LADEE (Lunar Atmosphere And Dust Environment Explorer) mission [7], 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 [32,36]. 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 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. One can configure MESA to specify how to check a property in a distributed manner. We present a case study demonstrating the impact of using concurrent monitors together with indexing. In this case study it is flight identifiers that are used as slicing criteria. We evaluate how different concurrency strategies impact the performance. The results are positive, demonstrating that concurrency used to handle slices of a trace can be beneficial. This is not a completely obvious result considering the cost of scheduling threads for small tasks. The main contribution of the paper is providing an extensive empirical assessment of asynchronous concurrent monitors implemented as actors. The paper also presents a new runtime verification tool, MESA, and its application on a real case study.

2 Related Work

Amongst the most relevant work is that of Basin et. al. [8]. 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 slices as there are monitors. The individual monitors are considered black boxes, which can host any monitoring system fitting the expected monitor interface. Another attempt in a similar direction is that of Hallé et. al. [18], which also submits trace slices to parallel monitors, a development of the author's previous work on using MapReduce for the same problem [5]. Reger in his MSc dissertation [35] experi-

mented with similar ideas, creating parallel monitors to monitor subsets of value ranges. However, in that early work the results were not promising, possibly due to the less mature state of support for parallelism in Java and hardware at the time. Berkovich et. al. [9] 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 in different computing hardware than the execution of the system under observation. Francalanza and Seychell [17] 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 [13] review different approaches to monitoring multi-threaded Java programs, which differs in perspective from the monitoring system itself to be parallel. Francalanza et. al. [16] 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.

The work by Burlò et. al. [10] targets open distributed systems and relies on session types for verification 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. The work by Neykova and Yoshida [33] applies runtime verification to ensure a *sound* recovery of distributed Erlang processes after a failure occurs. Their approach is based on session types that enforce protocol conformance. In [28], 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, as our approach, provide a temporal logic API for specifying properties, which is argued to be an advantage. Daut as well as TraceContract 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 manner. This, however, requires that the worker monitors do not rely on an internal state representing a summary of past events. The work by Attard and Francalanza [3] 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 system under observation. 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.

3 An Overview of 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 Figure 1. A MESA system

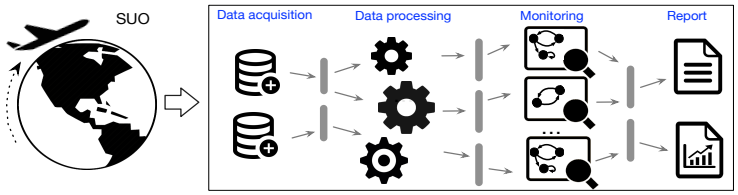


Fig. 1: Overview of a MESA actor-based monitoring system.

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 [34].

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 the 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 monitoring system (Figure 1) one needs to specify the actors and the way they are connected with communication channels in a HOCON 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 in the process of getting open-sourced. Akka provides the actor model implementation. RACE, built on top of Akka, is mainly 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. 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.

TraceContract and Daut are trace analysis DSLs where given a program trace and a formalized property, they determine whether the property holds for the trace. `Monitor` is a main class in these DSLs which encapsulates property specification capabilities and implements a key method, `verify`, that for each incoming event updates the state of the monitor accordingly. Instances of `Monitor` are referred to as monitors from here on. Similar to actors `receiveLive` method, `Monitor.verify` 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 the event. Using indexing, a Daut monitor only iterates over an indexed subset of states.

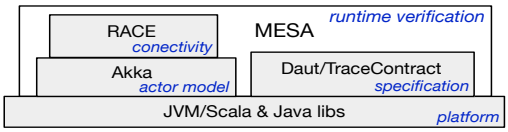


Fig. 2: The MESA framework infrastructure.

Properties in MESA are defined as subclasses of `Monitor`. The actors in the monitoring step (Figure 1), referred to as *monitor actors*, hold an instance 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 by data parametrized events

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. 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 that merge together, and 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. STARs are specfically designed for flights operated under Instrument Flight Rules under which the aircraft is navigated by reference to the instruments in the aircraft cockpit rather than using visual references. STAR routes can only be used by aircrafts equipped with a specific navigation system called RNAV.

One of the ongoing focus points of the FAA is to increase the utilization of STAR procedures. From 2009 to 2016, as part of the NextGen project, 264 more STAR procedures were implemented on an expedited timeline [41] which led to safety concerns raised by airlines and air traffic controllers including numerous unintentional pilot deviations [12,24]. 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 [40] studies RNAV STAR adherence trends based on a data mining methodology, and shows deviation patterns at major airports [4].

The case study applies runtime verification to check if flights are compliant with the designated STAR routes in realtime. A navigation specification for flights assigned to a STAR requires a lateral navigation accuracy of 1 NM[3] for at least 95% of the flight time [25]. 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 [40].

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

4.1 Formalizing Property P_{RSA}

For a sake of brevity, we say a flight *visits* a waypoint if the flight crosses inside a 1.0 NM radius around the waypoint. We say an event occurs when the aircraft under scrutiny visits a waypoint that belongs to its designated STAR route. For example, in Figure 3, 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 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 in L it returns the subsequent

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

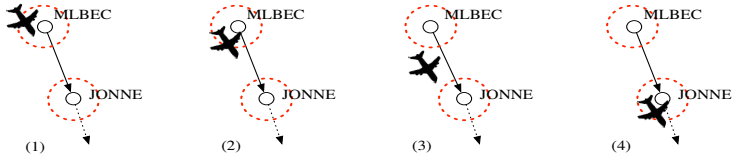


Fig. 3: The sequence of events for the aircraft is MLBEC MLBEC JONNE.

waypoint in the route. For example, $next$(MLBEC) returns JONNE (Figure 3). The finite state machine for Property P_{RSA} is the tuple $(Q, \Sigma, q_0, F, \delta)$ where

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

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 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.

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

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

All the events and types encapsulated by them are implemented as Scala `case` classes due to their concise syntax and built-in pattern matching support that

```
1  class P_RSA(config: Config) extends daut.Monitor(config) {
2    always {
3      case e@Visit(info@Info(_, track: Track), wp: Waypoint) if (isValid(e)) =>
4        if(wp == first) nextState(wp, track.cs) else error(msg)
5      case e@Completed(ti) if (isValid(e)) => error(msg)
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`, _,_,_), track:Track),`next`) =>
13         if(next == last) accept(last, track) else nextState(next, cs)
14       case Visit(Info(State(`cs`, _,_,_),_),_) => error(msg)
15       case Completed(Track(`cs`,_,_)) => error(msg)
16       case StarChanged(Track(`cs`,_,_)) => drop(cs)
17     }
18   }
19   def accept(wp: Waypoint, track: Track): state = {report(msg);ok}
20   def error(msg: String): state = {report(msg);err}
21   def drop(cs: String): state = {dropMonitor(cs);ok} ...
22 }
```

Fig. 4: Implementation of Property P_{RSA}.

facilitates the implementation of data-parametrized state machines. The class `Visit` represents an event where the flight visits the given waypoint, `Waypoint`. `Completed` indicates that the flight is completed. `StarChanged` indicates that the given flight is assigned to a new STAR route.

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

We implement P_{RSA} as a Daut monitor (Figure 4). A Daut monitor maintains the set of all *active* states representing the current states of the state machines. For each incoming event, new target states of transitions may be created and the set of active states updated. A state is presented by an object of type `state`, and the set of transitions out of the state is presented by an instance of `Transitions`, which is a partial function of type `PartialFunction[E,Set[state]]` where `E` is a monitor type parameter.

The functions `always` and `watch` act as states. They accept as argument a partial function of type `Transitions` and return a `state` object. The `case` statements, representing transitions at states, are matched against incoming events. The verification starts from the state `always`, and `watch` represents a state at which the flight visits a waypoint. For `case` statements in `always`, `isValid` is used as a pattern guard to narrow down the set of events triggered on the state

to the ones relevant to the STAR routes assigned to the monitor. Moreover, since `always` is always an active state, `isValid` ensures that only one event per flight is triggered on `always`. The first `case` in `always` matches against the event `Visit` where a flight visits a waypoint. Then, it checks if the waypoint, `wp`, visited by the flight is the first waypoint of the route. If so, the method `nextState` is invoked which advances the state to `watch`. Otherwise, the method `error` advances the state to `err`. The second `case` matches against `Completed`, indicating that the flight is completed without visiting any waypoints, and invokes `error`.

The `watch` state waits to receive an event that triggers one of its transitions and then leaves the state. The input parameters of `nextState`, `wp` and `cs`, represent the waypoint being visited by the flight and the flight call sign. To ensure that only events associated with this flight can match against `case` statements, all the patterns match the call sign for the incoming event, `cs`, against the value of the flight call sign. This is done by using back-quotes for associated parameter in the typed patterns, `cs`. The first `case` in `watch` matches against the event where the flight visits the current waypoint, and calls `nextState(wp,cs)` to remain in the current state. The variable `next` is set to the next waypoint in the STAR route. The second `case` matches against the event where the flight visits the waypoint `next`. It checks if `next` is the last waypoint, and if so, it calls `accept` which returns the object `ok`, representing the accepting state. If `next` is not the last waypoint, it calls `nextState(next,cs)` to advance to the state corresponding to `next`. Next `case` matches against `Completed` which calls `error` to advance to the `err` state. Finally, last `case` matches against `StarChanged` which calls `drop` to discard the analysis for the flight.

4.3 A MESA Monitoring System for P_{RSA}

Figure 5 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) [19]. SWIM implements a set of information technology principles in NAS which consolidates data from many different 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 [37] 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 realtime fight data from a SWIM service, SFDPS (SWIM Flight Data Publication Service)[15]. `SFDPS-importer` publishes the data to the channel `sfdps`.

The data processing step parses the SFDPS data obtained from the previous stage and generates a trace, composed of event objects, relevant to the property. This done via a pipeline of actors that parse the SFDPS messages in XML (`sfdps2track` and `sfdps2state`), filter irrelevant data (`filter`), and finally

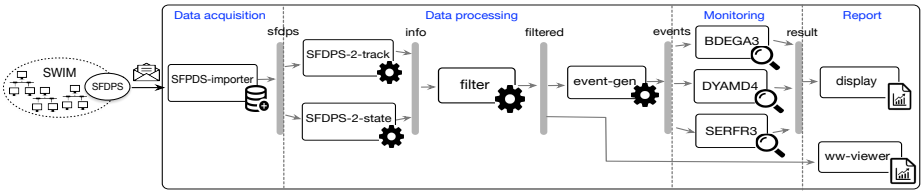


Fig. 5: A MESA instance for verifying Property P_{RSA} for STARs at SFO.



Fig. 6: Flight deviation from assigned RNAV STARs detected at SFO.

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` (Figure 4). They subscribe to the channel `trace`, and feed their underlying `P_RSA` object with incoming events. Each monitor actor in Figure 5 is associated to a STAR procedure at SFO which checks for the flights assigned to that 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 to provide interactive geospatial visualization of flight trajectories.

Using the MESA system shown in Figure 5, we discovered violations of P_{RSA}. Figure 6 includes snapshots from our visualization illustrating two cases where P_{RSA} was violated. It shows that the flight United 1738 missed the waypoint LOZIT, and the flight Jazz Air 743 missed the initial waypoint BGGLO.

5 Experiments

This section presents our experiments evaluating the impact of using concurrent monitors and indexing. More details on the experiments can be found in [39]. The experiments uses 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. We use the state of

flights as events captured by `State` instances, and specify the property p as a data-parameterized finite state machine using Daut as follows, where t1 and t2 represent the event time.

always {**case** State(cs,_,_,t1)=>watch {**case** State('cs',_,_,t2)=>t2.isAfter(t1)}}

This property 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` which maintains a list of monitor instances, all capturing the same p, as its sub-monitors.

We evaluate the impact of concurrency in the context of indexing. Indexing can be applied both at the monitor level or the dispatcher level. Indexing at the monitor level is supplied by Daut. We activate this feature by implementing an indexing function in the Daut monitor that uses the call signs carried by events to retrieve the set of relevant states for analysis 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.

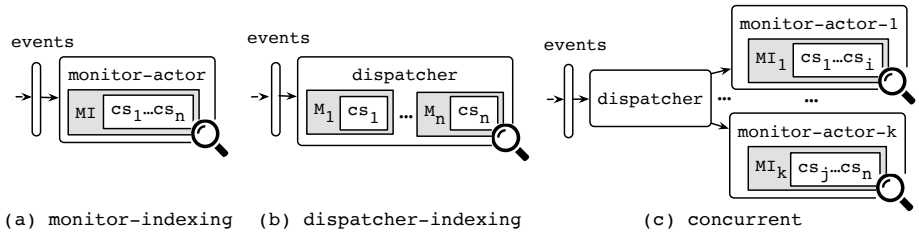


Fig. 7: Actor-based monitoring systems used in the experiment.

5.1 Monitoring Systems

The experiments use four different MESA systems which are only different in their monitoring step. They all use the same actors to extract the recorded SFDPS data, generate a trace composed of `State` objects, and publish the trace to a channel, `events`, accessed in the monitoring step. The monitoring step for each system is illustrated in Figure 7. 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, the features of which are summarized in Table 8.

– `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`. In a way, the monitoring step of this configuration is equivalent to directly using the Daut tool to process the trace sequentially.

- `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. 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. By setting a limit, one monitor actor could be assigned to more than one call sign. The latter 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.

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

Fig. 8: The main features of the monitoring systems presented in Figure 7.

5.2 System Setup

All experiments are 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, 3.30GHz 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 Figure 7 is 3215. The number of sub-monitors in `P_SEQ` is set to 2000. The Java heap size is set to 12 GB. Our experiment uses a default setting of the Akka scheduler which associates all actors to a single thread pool with 60 threads, and uses the default value 5 for actors *throughput*, the maximum number of messages processed by the actor before the assigned thread is returned to the pool.

5.3 Evaluation

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 7 compares the run times for the monitoring systems presented in Figure 9. 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 `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`. Considering the similarity between their indexing mechanisms, the difference mostly amounts to the implementation. 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 (Figure 10). The CPU utilization for `monitor-indexing` is mostly under 30% and for `dispatcher-indexing` is mostly between 40% and 50%. For `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 system exhibit a similar heap usage which mostly remains under 10G.

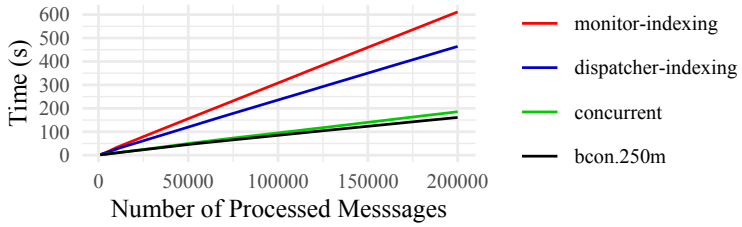
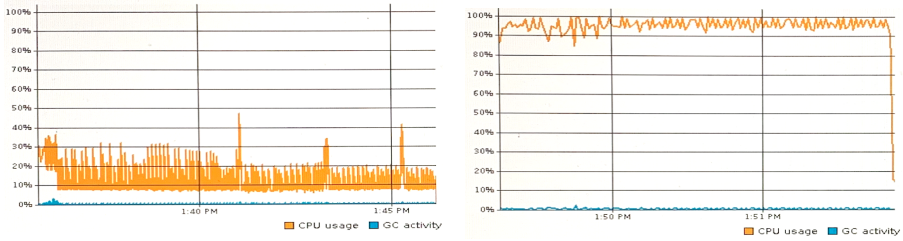


Fig. 9: Comparing the run times of different MESA actor systems.

Figure 9 shows that limiting the concurrent monitors to 250 results in a better performance than using one monitor actor per flight in `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 Table 11. The system performs best with 250 monitor actors, and from there



(a) `monitor-indexing` (b) `bounded-concurrent`(250 monitors)

Fig. 10: The CPU utilization profiles obtained by VisualVM.

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 could lead to overhead, e.g. it leads to 11% overhead when using 3215 monitor actors, and improves the performance by 45% when using 125 monitor actors.

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

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

5.4 Actor Parameter Evaluation

We also evaluate performance parameters for individual dispatcher and monitor actors in each monitoring system, including the average service time, and the average wait time for messages in the mailbox. The relevant 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 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 invocation, and publishes a data container with the recorded times to the channel accessed by `stat-collector`. To obtain information from actor mailboxes, we implement a new mailbox 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 publishes a data container with the recorded times to the channel accessed by `stat-collector`. Any MESA actor can be configured to use these features, referred as ASF. 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 increases as the number of monitor actors increases.

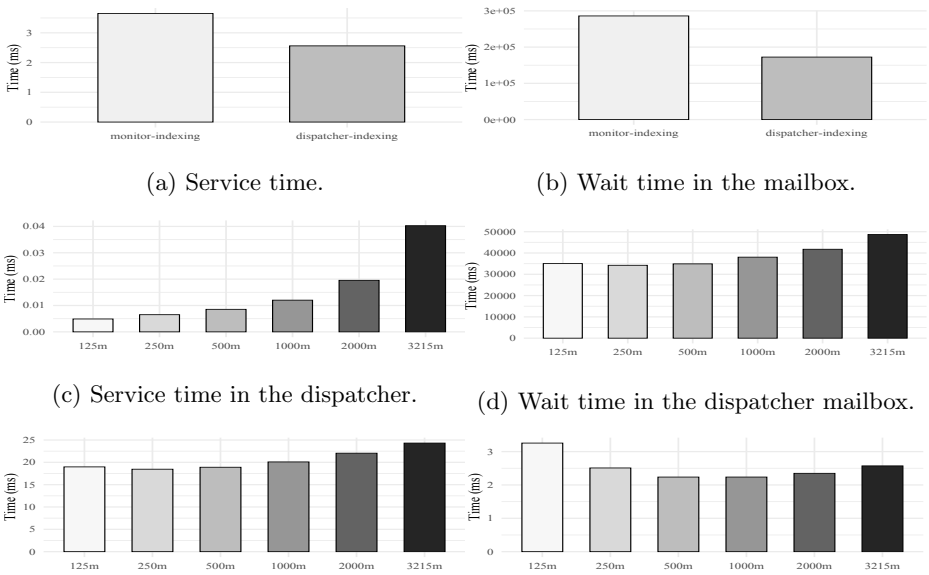
Figure 12 compares the performance parameters for individual actors. Figure 12a and 12b show that the monitor actor in `monitor-indexing` has a longer service time and longer wait time in the mailbox comparing to the dispatcher in `dispatcher-indexing`. Figure 12c and 12d compare the dispatcher performance metrics for `bounded-concurrent` with different numbers of monitor actors. Figure 12e and 12f present the monitor actors performance metrics for the same systems. The average service time for the dispatcher and monitor actors increases as the number of actors increases. Increasing the monitor actors increases the load on the dispatcher since it needs to generate more monitor actors. Decreasing the number of monitor actors increases the load on individual monitor actors since each actor monitors more flights. On the other hand, applying indexing within the monitor actors improves their performance, however for monitors that track small number of flights, indexing can lead to overhead leading to longer service times.

The message wait time in the dispatcher mailbox increases as the number of actors increases (Figure 12d). In general, with a constant thread pool size, increasing actors in the system can increase the wait for actors to get scheduled, leading to longer wait for messages in mailboxes. However, in the case of monitor actors the mailbox wait is longer with smaller number of actors (Figure 12f). This is due to higher arrival rate of messages in these systems since each monitor actor is assigned to higher number of flights.

6 Discussion

Applying MESA on NAS demonstrates that our approach can be used to effectively detect violations of temporal properties in a distributed SUO. We show the impact of using concurrent monitors for verification. Our evaluation includes a setting that resembles using an existing trace analysis tool, Daut, directly. Comparing this setting to the concurrent monitoring setting reveals that employing concurrent actors can considerably improve the performance. MESA is highly extensible, and provides flexibility in terms of incorporating new DSLs. It can be viewed as a tool that provides concurrent monitoring platform for existing trace analysis DSLs.

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 cannot be determined 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 multi-core machine can



(a) Service time.

(b) Wait time in the mailbox.

(c) Service time in the dispatcher.

(d) Wait time in the dispatcher mailbox.

(e) Service time in the monitor actor.

(f) Wait time in the monitor actor mailbox.

Fig. 12: Comparing the monitors performance metrics for MESA systems.

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. MESA is a highly configurable platform that 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.

As shown in Figure 2, our framework runs on top of 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 microbenchmarking on JVM which is a well-known problem. Certain characteristics of 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 [26].

Several of the mentioned works [8,18,9,3], support the observation that concurrency in one form or other, using asynchronous message passing, can improve performance of runtime verification systems. The works most relevant to our combination of slicing and concurrency are [8,18]. Basin et. al. [8] provide performance results for the use of slicing together with concurrency, but do not

compare these with runs without concurrency. However, the logs analyzed contain billions of events, supporting the observation that exactly this use of concurrency is performance enhancing. Hallé et. al. [18] do not provide performance results for specifically the combination of slicing and concurrency.

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 difference 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.

7 Conclusion

In this work we have presented a runtime verification tool that employs concurrent monitors as actors. Our approach allows for specifying properties in data-parameterized temporal logic and state machines, provided by existing trace analysis DSLs. We present a case study demonstrating how the tool is used to obtain live air traffic data feeds and verify a property that checks if flights adhere to assigned arrival procedures. We evaluate different combinations of indexing and concurrency, and observe 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.

References

1. Akka (2020), `http://doc.akka.io/docs/akka/current/scala.html`
2. 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 International Publishing (2015). https://doi.org/10.1007/978-3-319-25423-4_1
3. 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 (2017). https://doi.org/10.1007/978-3-319-66197-1_14
4. Avrekh, I., Matthews, B.L., Stewart, M.: RNAV Adherence Data Integration System Using Aviation and Environmental Sources. Tech. rep., NASA Ames Research Center (6 2018)
5. 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 Heidelberg, Berlin, Heidelberg (2013)
6. 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, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_7

7. Barringer, H., Havelund, K., Kurklu, E., Morris, R.: Checking Flight Rules with TraceContract: Application of a Scala DSL for Trace Analysis. Tech. rep., Jet Propulsion Laboratory, National Aeronautics and Space Administration, Pasadena, California (2011), http://hdl.handle.net/2014/42194
8. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specification. Formal Methods in System Design **49**, 75–108 (2016). https://doi.org/10.1007/s10703-016-0242-y
9. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime verification with minimal intrusion through parallelism. Formal Methods in System Design **46**, 317–348 (2015). https://doi.org/10.1007/s10703-015-0226-3
10. Burlò, C.B., Francalanza, A., Scalas, A.: Towards a Hybrid Verification Methodology for Communication Protocols (Short Paper). In: Gotsman, A., Sokolova, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. Lecture Notes in Computer Science, vol. 12136, pp. 227–235. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_13
11. 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, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_15
12. Department of Transportation, Federal Aviation Administration: Implementation of Descend Via into Boston Terminal area from Boston ARTCC (2015)
13. 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
14. 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 (01 2013). https://doi.org/10.3233/978-1-61499-207-3-141
15. SWIM Flight Data Publication Service (2020), https://www.faa.gov/air_traffic/technology/swim/sfdps/
16. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems, pp. 176–210. Springer International Publishing, cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
17. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Formal Methods in System Design **46**(3), 226–261 (2015). https://doi.org/10.1007/s10703-014-0217-9
18. 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
19. Harris Corporation: FAA Telecommunications Infrastructure NEMS User Guide (2013)
20. Havelund, K.: Data Automata in Scala. In: Symposium on Theoretical Aspects of Software Engineering Conference. pp. 1–9. Changsha, China (2014). https://doi.org/10.1109/TASE.2014.37
21. Havelund, K.: Daut (2020), https://github.com/havelund/daut
22. Havelund, K.: TraceContract (2020), https://github.com/havelund/tracecontract

23. 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. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)
24. International Air Line Pilots Associations: FAA Suspends OPD Arrivals for Atlanta International Airport (2016)
25. International Civil Aviation Organization (ICAO): Performance-based Navigation (PBN) Manual, 3 edn. (5 2008)
26. JMH - Java Microbenchmark Harness (2020), `https://openjdk.java.net/projects/code-tools/jmh/`
27. Joyce, J., Lomow, G., Slind, K., Unger, B.: Monitoring Distributed Systems. ACM Transactions on Computer Systems **5**(2), 121–150 (1987). https://doi.org/10.1145/13677.22723
28. 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 Computer Society (2017)
29. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
30. Mehlitz, P.: RACE (2020), `http://nasarace.github.io/race/`
31. 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 (09 2016). https://doi.org/10.1109/DASC.2016.7777991
32. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer (2011)
33. 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 (2017). https://doi.org/10.1145/3033019.3033031
34. Rasmussen, S., Kingston, D., Humphrey, L.: A Brief Introduction to Unmanned Systems Autonomy Services (UxAS). 2018 International Conference on Unmanned Aircraft Systems (ICUAS) pp. 257–268 (2018). https://doi.org/10.1109/ICUAS.2018.8453287
35. Reger, G.: Rule-Based Runtime Verification in a Multicore System Setting. Master's thesis, University of Manchester (2010)
36. 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, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
37. Richards, M., Monson-Haefel, R., Chappell, D.A.: Java Message Service. O'Reilly Media, Inc., 2nd edn. (2009)
38. Roestenburg, R., Bakker, R., Williams, R.: Akka in Action. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2015)
39. Shafiei, N., Havelund, K., Mehlitz, P.: Empirical Study of Actor-based Runtime Verification. Tech. rep., NASA Ames Research Center (6 2020)
40. Stewart, M., Matthews, B.: Objective assessment method for RNAV STAR adherence. In: DASC: Digital Avionics Systems Conference (2017)
41. U.S. Department of Transportation. Federal Aviation Administration: Performance Based Navigation PBN NAS Navigation Strategy (2016)