MESA: Message-Based System Analysis Using Runtime Verification

Nastaran Shafiei, Oksana Tkachuk, Peter Mehlitz

SGT, Inc./NASA Ames Research Center
Moffett Field, CA, USA

Abstract. In this paper, we present a novel approach and framework for run-time verification of large, safety critical messaging systems. This work was motivated by verifying the System Wide Information Management (SWIM) project of the Federal Aviation Administration (FAA). SWIM provides live air traffic, site and weather data streams for the whole National Airspace System (NAS), which can easily amount to several hundred messages per second. Such safety critical systems cannot be instrumented, therefore, verification and monitoring has to happen using a nonintrusive approach, by connecting to a variety of network interfaces. Due to a large number of potential properties to check, the verification framework needs to support efficient formulation of properties with a suitable Domain Specific Language (DSL). Our approach is to utilize a distributed system that is geared towards connectivity and scalability and interface it at the message queue level to a powerful verification engine. We implemented our approach in the tool called MESA: Message-Based System Analysis, which leverages the open source projects RACE and TraceContract. RACE is a platform for instantiating and running highly concurrent and distributed systems and enables connectivity to SWIM and scalability. TraceContract is a runtime verification tool that allows for checking traces against properties specified in a powerful DSL. We applied our approach to verify a SWIM service against several requirements. We found errors such as duplicate and out-of-order messages.

1 Introduction

Message-based system is a system that contains multiple components that primarily communicate by exchanging messages over networks. Safety-critical systems may choose to use a message-based architecture in order to take advantage of parallelism while avoiding the problems inherent to shared state. Due to the rise of safety-critical message-based systems, there is a need for formal methods to verify the quality of such systems.

One example of a safety-critical, message-based system is the Next Generation Air Transportation System (NextGen) [12]. The data-sharing backbone of NextGen is the System Wide Information Management (SWIM), which provides users with sensitive live data feed such as en route flight data. SWIM is a highly distributed system that consolidates data from many sources. It also deals with

a large amount of data. Our experiments show that the number of messages received for en route flights varies between 70 and 500 messages per second.

One of our projects at NASA relies on data feeds from SWIM, and we observed that flight data obtained from SWIM exhibited incorrect message sequences. This motivated us to closely look into the trace analysis problem (runtime verification) for safety-critical message-based systems. Section 2 explains the SWIM system and the problems we observed in more detail.

Our ultimate goal is to provide a tool that extracts sequences of messages from a message-based System Under Test (SUT) and checks them on-the-fly against temporal properties specified using an expressive specification language. Dealing with safety-critical systems requires a nonintrusive approach, since the source code from these systems is often not available. Even when the source is available, any potential malfunction that may be introduced by instrumentation cannot be tolerated. Another challenge is scalability, since the system, especially the ones producing live feeds, often deal with large messages received in high volumes.

In this paper, we introduce MESA, a tool written in the Scala programming language. Its main feature is to separate data acquisition, post-processing, property specification, and runtime verification into dedicated system components. The data acquisition component provides the SUT connectivity, the post-processing ensures scalability, and provides the variables for a type-checked property specification language. MESA is built upon two existing tools: RACE [13] and TraceContract [9].

RACE, Runtime for Airspace Concept Evaluation, is designed as a framework to build airspace simulations. Airspace simulations typically involve components that need to incorporate external systems, e.g., live data feeds (servers, sensors, aircraft), hardware simulators, and data distribution services (bus systems). RACE provides connectivity to these systems. The architecture of RACE, explained in Section 3.1, is highly extensible, and provides scalability for the high volume of messages we see in SWIM.

TraceContract is a runtime verifier that provides its own property specification language in form of an internal DSL that covers LTL [15] formulas and state machines. TraceContract previously has been used for the analysis of command sequences against flight rules [10] for the NASA LADEE (Lunar Atmosphere And Dust Environment Explorer) mission [3]. In MESA, we use TraceContract to specify properties on sequences of messages produced by the SUT. Section 3.2 describes TraceContract in more details.

This paper makes the following contributions: (1) identification of the challenges involved in runtime verification of systems like SWIM, (2) implementation of solutions to these challenges in a framework, MESA, described in Section 4 and (3) application of MESA in order to identify errors in SWIM sequences of en route flight data messages, detailed in Section 6.

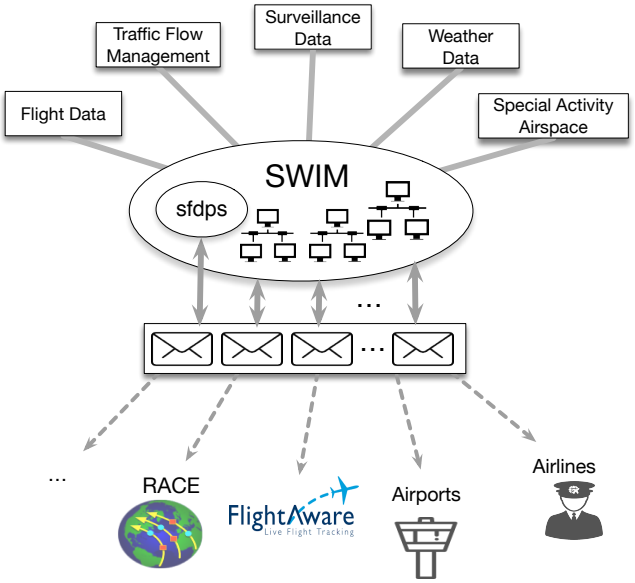

Fig. 1. SWIM Overview

2 Motivating Example

SWIM is considered a key part of the NAS for the NextGen air transportation systems [7]. It implements a set of information technology principles. It collects data from many different facilities and resources, and combines them into data feeds to provide authorized users with relevant and comprehensible information. As shown in Figure 1, the information provided by SWIM includes flight data, weather information, surveillance data, airport operational status, etc. Some of the SWIM users are airlines, airports, external applications such as FlightAware [2] and RACE. SWIM provides authorized users with access to the NAS data through publish/subscribe information exchange using a Java Message Service (JMS) implementation. JMS is a messaging API that allows for asynchronous sending and receiving of messages. The SWIM messages published in the JMS server are in standard XML formats.

SWIM is a highly distributed system. Its implementation follows a service-oriented architecture which is a collection of services that communicate with one another. In this work, we focus on verifying the SWIM Flight Data Publication Service (SFDPS). This service publishes en route flight data from 20 different FAA air traffic control systems. At any given day, there can be more than 4500 simultaneous flights in the US airspace, each of them updated from various input sources in a 12 seconds interval.

When visualizing en route flight data using RACE, we observed traces that contain incorrect patterns. Traces that include duplicated messages which are

either lexically duplicated or encapsulate the same flight information, including the same aviation call sign (unique identifiers assigned to aircraft), position, and time are considered incorrect. Moreover, en route flight messages have a time stamp attached to them, and messages associated with a call sign should be ordered by their time stamps. Examining traces obtained from the SFDPS service revealed traces with duplicated and out-of-order messages.

Our aim is to identify these faulty patterns on-the-fly. Motivated by this example, we present our work that applies trace analysis against properties formalized in LTL and finite state machines.

3 Background

We now discuss the primary features of RACE and TraceContract that MESA utilizes.

3.1 RACE

RACE [13] is a platform for instantiating and running highly concurrent and distributed systems. RACE employs the actor programming model, as implemented in the Akka [1] framework. Akka actors communicate through asynchronous messages and do not share state. Each actor is associated with a mailbox and processes its messages sequentially. RACE is implemented in the Scala [6] programming language, which improves type safety compared to other JVM languages.

RACE is a highly-configurable and extensible platform, which makes it suitable for a wide range of applications. Specifically, it can be used to rapidly build simulations that span several machines (including synchronized displays), interface existing hardware simulators and other live data feeds, and incorporate sophisticated visualization components. RACE includes many building blocks to create distributed systems, including actors to import, export, translate, filter, record, archive, replay, and visualize data.

In this paper, we use RACE to connect to the SWIM server and to filter its messages based on the topics of interest. Figure 2 shows a RACE sample configuration file that enables this use case. The configuration, written in JSON, specifies the actors to instantiate (using keywords `name` and `class`), the channels the actors use to read or write data to, as well as topics of interest. The example configuration in Figure 2 tells RACE to instantiate 3 actors: (1) `JMSImportActor` to import the data of specific topic, `sfdps`, to channel `/swim`, (2) `TranslatorActor` to translate data from XML to objects, and finally (3) `ProbeActor` to receive the resulting data. Note that the `JMSImportActor` makes use of the user and password information, which is encrypted in this case (indicated by `??`). The number and types of actors the user needs to specify depends on the task.

RACE is developed at NASA Ames Research Center. It is open sourced under the Apache v2 license and is available at [5].

```
actors = [
  { name = "jmsImporter"
    class = ".jms.JMSImportActor"
    broker-uri = "tcp://localhost:61616"
    user = "??swim.user"
    pw = "??swim.pw"
    write-to = "/swim"
    jms-topic = "nasa.topic05_12.sfdps"
  },
  { name = "fixm2fpos"
    class = ".actor.TranslatorActor"
    read-from = "/swim"
    write-to = "/fpos"
    translator = { class = ".air.translator.FIXM2FlightObject"}
  },
  { name = "fposReceiver"
    class = ".actor.ProbeActor"
    read-from = "/fpos"
  }]
```

Fig. 2. RACE Configuration

3.2 TraceContract

TraceContract [9] is an API for performing trace analysis. Given a program trace and a formalized property, trace analysis checks whether the property holds for the trace. TraceContract is implemented in Scala and takes advantage of Scala's support for Domain Specific Languages (DSLs). TraceContract is an internal (embedded) DSL, that combines DSL with all of Scala's features, and supports specification of properties as a combination of data parameterized state machines and temporal logic.

Figure 3 shows APIs provided by TraceContract. Given a trace as a finite sequence of `Events` and a `Formula`, TraceContract checks whether the formula holds true for the trace. The two main classes, `Monitor` and `Formula` provide APIs for writing temporal properties (e.g., `globally`, `eventually`) and state machines (e.g., `state`). For example, consider the following property: a SWIM server feed should not contain duplicate messages. In other words, it should always hold that for any message that the receiver actor dequeues from its mailbox, if the actor sees the same message again, an error should be raised. This property can be formalized using the following specification in TraceContract:

```
property ('lex_dup){
  always{
    case (Dequeue(msg)) =>
      state{
        case Dequeue('msg') => error
      }
```

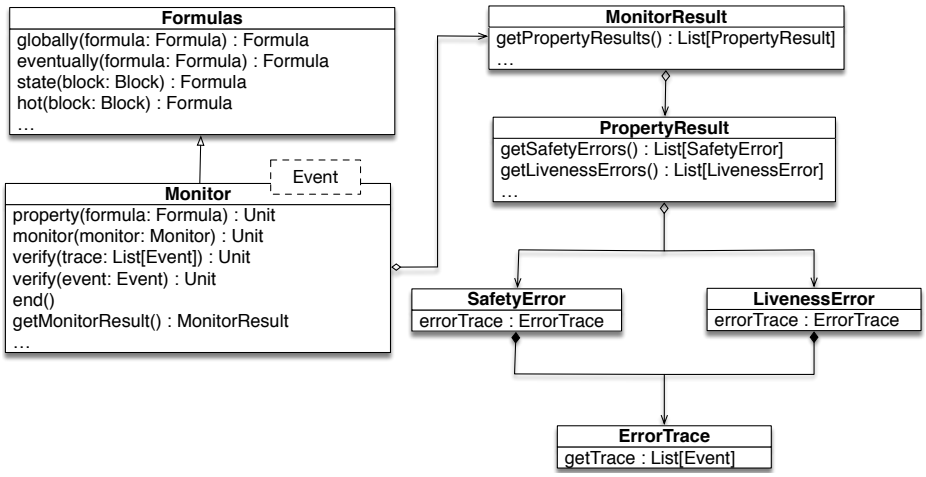

Fig. 3. TraceContract classes

```
  }
}
```

In section 6, we present a list of properties we checked for SWIM. TraceContract is developed at NASA's Jet Propulsion Laboratory (JPL) and is currently in the process for an Apache license open source release.

4 Approach

MESA performs trace analysis on sequences of messages. The approach is non-invasive, and applies runtime verification to check for temporal properties that are specified in LTL or finite state machines. The high-level view of the approach is shown in Figure 4. The two main functionalities of MESA are (1) extracting sequences of messages from the messaging system construct of the SUT, and (2) verifying them against specified temporal properties. MESA utilizes the tools RACE and TraceContract, respectively, to provide these functionalities.

RACE provides dedicated actors, referred to as importers, that can subscribe to commonly-used messaging system constructs, such as JMS server, Kafka, and DDS. To connect to a messaging system, a user needs to specify the respective importer in the RACE configuration, as shown in Figure 2.

In order to receive message sequences from RACE, we added a functionality in MESA that allows for retrieving messages from individual actors' mailboxes. User can specify which actors' mailboxes are accessed from MESA to extract sequences of messages. This functionality is provided using the Akka extension mechanism, which allows for replacing core classes in Akka. MESA replaces the type representing actors mailboxes in Akka with a new implementation. The new implementation extends the mailbox implementation with two trigger points that

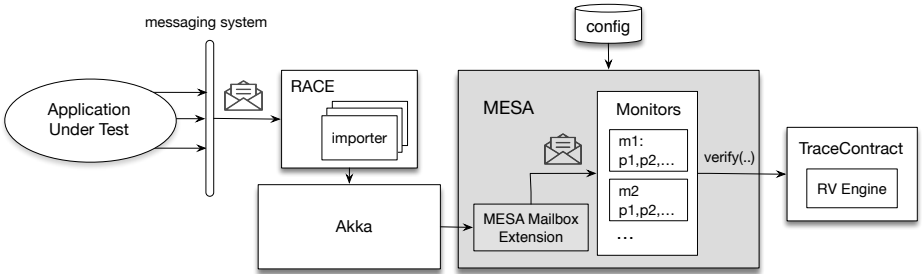

Fig. 4. MESA Architecture

```
akka {
  extensions = ["traceExtraction.MesaExtensionImp"]
}
mesa-mailbox {
  mailbox-type = "traceExtraction.MesaMailboxType"
}
akka.actor.deployment {
  fposReceiver {
    mailbox = mesa-mailbox
  }
}
monitor {
  class = "SwimSFDPSMonitor"
}
```

Fig. 5. MESA Configuration

generate events upon enqueueing and dequeueing messages. A message that is enqueued to or dequeue from a mailbox is wrapped into an object representing the event, and it is passed over to the corresponding monitor object in MESA. We consider two types of events that capture enqueueing and dequeueing messages. In Section 5, we elaborate on the types of these events.

The configuration in Figure 5, written in JSON, shows how MESA can be configured to access the mailboxes of RACE actors. `mesa-mailbox` represents the mailbox extension implemented by the class `traceExtraction.MesaMailboxType`. Moreover, the configuration in Figure 5 sets the actor with the name `fposReceiver` to use an instance of `traceExtraction.MesaMailboxType` as its mailbox.

Using configuration, the user can also specify the monitors to be used for verification. Monitors in MESA are instances of the class `MesaMonitor`. As it can be seen in Figure 6, this class extends the class `Monitor` in TraceContract. `Monitor` provides methods needed for writing properties. `MesaMonitor` extends this class with a configuration mechanism and methods to identify end of traces, collect statistics about messages, instantiate monitors on-the-fly, etc. `Monitor` is param-

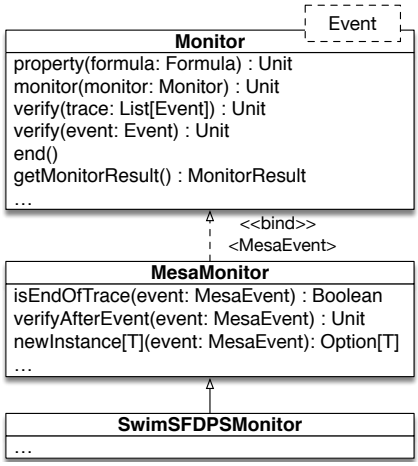

Fig. 6. MESA monitors type

eterized with the type of the event used when defining properties. `MesaMonitor` binds the event type with `MesaEvent` described in the next section.

The MESA monitors which represent properties over sequences of messages must extend the class `MesaMonitor`. For example, in Figure 6, the monitor `SwimSFDPSMonitor` represents properties defined for message sequences obtained from the SFDPS service. The last section in Figure 5 shows how MESA can be configured to use the `SwimSFDPSMonitor` monitor for every actor that uses the `mesa-mailbox` extension.

As part of the mailbox extension initialization, the monitors associated with the actor are created. At runtime, the actor passes events encapsulating messages to the monitors by calling the method `verify(event)` on the monitor objects. After receiving every event, the properties in the body of the monitor are checked and violations, if any, are reported.

In the next section, we explains how the properties for the SFDPS service are formalized in the body of the `SwimSFDPSMonitor` monitor.

5 Formalizing Properties

Table 1 outlines the properties we used for our evaluation along with their natural language descriptions. In this section, we describe the formalization of these properties. We specify the properties using the state machine API of TraceContract. The properties are defined over the sequences of messages retrieved from the SFDPS service. Note that, for the sake of readability, we present slightly simplified versions of the formalizations used in our experiments.

Before formalizing the properties, we specify the events. The events used in our system are defined below.

pattern	description
lex_dup	messages must not be lexically identical
flight_info_dup	messages must not encapsulate the duplicate flight data including the same call sign, position, altitude, and time
flight_seq_order	messages with the same call sign must be ordered by their time tags

Table 1. Properties and their natural language descriptions.

```
abstract class MesaEvent
case class Enqueue(message: Any) extends MesaEvent
case class Dequeue(message: Any) extends MesaEvent
```

The abstract type `MesaEvent` represents the event objects. Two types of events are defined by `Enqueue` and `Dequeue` classes which are subclasses of `MesaEvent`. These classes are defined as `case` classes. `case` classes allow for pattern matching over their constructor parameters. Traces analyzed by our approach are essentially sequences of `MesaEvent` instances.

A monitor maintains a list of formulas which are all evaluated after receiving each event. Properties in the body of monitors are presented by calls to the method `property` which has two arguments. The first argument is the name of the property, and the second argument is a `Formula` object.

The first property of the SFDPS service, named `lex_dup`, is formalized in Section 3.2. The method `always` represents a state, and given a `Block` instance, it returns a `Formula` object. `Block` represents a list of transitions leading out of the state. It is a partial function which is defined as follows:

```
type Block = PartialFunction[Event, Formula]
```

`always` is a state at which the monitor always waits for an event to match a transition in the `Block` domain. Once an event is matched, the formula returned by `Block` is added to the list of formulas of the monitor. For example, for the property `lex_dup`, the monitor always waits to observe the event `Dequeue`. When the incoming event matches `Dequeue` of some message, `msg`, `state` is added to the list of formulas. The method call `state` also represents a state, that given a `Block` object, returns `Formula`. `state` is a state at which the monitor waits an event to match a transition the `Block` domain. Once matched, the monitor moves to a new state which leads to `error` in this case which is also of type `Formula`. Note that the quoted variable names indicate that the current values must equal the value of the corresponding variables. For example, for the property `lex_dup`, the transition from `state` represents the `Dequeue` of the message `msg`.

The formalization of the second property, named `flight_info_dup`, is presented below. The message encapsulated by `Dequeue` must be of type `FlightPos`. The type `FlightPos`, which is defined in RACE, is used to encapsulate flight data. The `FlightPos` instances are generated by the `TranslatorActor` described in Section 3.1. The constructor parameters of `FlightPos` represent the flight

identifier, flight call sign, position, altitude, direction, and time associated with the message data, respectively.

```
property (‘flight_info_dup) {
 always {
  case Dequeue(FlightPos(_,cs,pos,alt,_,_,date)) =>
   state{
    case Dequeue(FlightPos(_,‘cs‘,‘pos‘,‘alt‘,_,_,‘date‘)) => false
   }
  }
 }
}
```

Note that underscore represents the wildcard pattern which is ignored. The property `flight_info_dup` states that it is always the case that, if a `Dequeue` of a `FlightPos` instance is observed, then the monitor advances to a new state, where if we observe `Dequeue` of a `FlightPos` instance with the same call sign, position, altitude, and time, it is an error.

The formalization of the next property, named `flight_seq_order`, is shown below.

```
property(’flight_seq_order) {
 always {
   case Dequeue(FlightPos(_, cs, _, _, _, _, date1)) =>
    state {
     case Dequeue(FlightPos(_, ‘cs‘, _, _, _, _, date2)) =>
            date2.isAfter(date1)
    }
  }
}
```

The property `flight_seq_order` states that it is always the case that, if `Dequeue` of a `FlightPos` instance is observed, then the monitor advances to a new state, where if `Dequeue` of a `FlightPos` instance, which includes the same call sign as the `FlightPos` instance seen in the previous state, is observed, it checks for the chronological order of the two `FlightPos` instances. If `date2` is strictly after `date1`, the monitor stops monitoring the formula, and it is removed from the list of formulas maintained by the monitor. Otherwise, it is an error, and the violation is reported.

In the following section, we present our results when checking sequences of messages against the properties presented in Table 1.

6 Evaluation

Our experiments include verifying sequences of messages obtained from the SFDPS service against the three properties presented in Table 1.

As mentioned earlier, our approach can be applied online by connecting RACE to the messaging system construct of the SUT. Our approach can be also applied offline by utilizing recorded data and importing them into RACE.

This can be accomplished by using `ReplayActor` in RACE. This actor publishes data from the given source to a given channel. We perform our experiments in the offline mode using `ReplayActor`. Figure 7 presents the configuration used to instantiate `ReplayActor` in the RACE environment. Replacing the `ReplayActor` configuration with the `JMSImportActor` configuration in Figure 2, RACE imports flight data from the file `sfdps.xml.gz` instead of directly connecting to the SWIM messaging system construct. `sfdps.xml.gz` includes flight data previously obtained from the SFDPS service.

```
actors = [
  { name = "sfdpsReplay"
    class = ".actor.ReplayActor"
    write-to = "/swim"
    pathname = "/sfdps.xml.gz"
    archive-reader = ".archive.TextArchiveReader"
  },
  ...]
```

Fig. 7. ReplayActor Configuration

All of our experiments are performed on a Mac OS X machine with a 2.8 GHz intel Core i7 processor and 16 GB 1600 MHz DDR3 memory. For these experiments, we use the Scala code version 2.12.1 and the Java HotSpot(TM) 64-Bit Server VM.

Table 2 presents our results. For each property, we include the results from traces of size 1000 and 2000 messages. Note that, for all the properties, we use the same sequence of SFDPS messages, and each property is evaluated in a separate experiment. The first column includes the name of the properties. Column `total time` presents the total time. Column `verification` presents the total time spent for evaluating formulas in monitors. For every message, this captures the time from the point that the message is received by a `MesaMonitor` instance until it is evaluated against the list of all formulas maintained by the monitor. The accumulation of these times for all messages is presented by Column `verification`. Column `event handling` represents the time obtained by subtracting the time in Column `verification` from the time in Column `total time`. This represents the time spent for extracting and handling events. Finally, the last column represents the number of violations of the respective property.

property	seq size	total time(s)	verification(s)	event handling(s)	# of violations
lex_dup	1000	140968	132320	8647	28
flight_info_dup	1000	79953	69861	10092	2
flight_seq_order	1000	77744	67650	10094	3
lex_dup	2000	1022483	1013524	8958	133
flight_info_dup	2000	544889	534485	10404	9
flight_seq_order	2000	527201	516865	10335	10

Table 2. RV results for the SFDPS service against the properties in Table 1

The results shows that that time used for verifying the property `lex_dup` (see the formalization in Section 3.2) is about 1.8 times higher than the other two properties. This property does not apply any pattern matching on the message structure, that is, every `Dequeue` event matches the transition leading out of the `always` state, and thus, leads to a new formula. The list of formulas to be evaluated after receiving every event grows faster for the property `lex_dup`. For example, comparing to the property `flight_info_dup`, where only the events of type `Dequeue` that encapsulate a message of type `FlightPos` match the transition leading out of the `always` state, and lead to a new formula.

The result also shows that as the size of the trace doubles, the total time for the property `lex_dup` becomes 7.2 times larger, and for the other two properties becomes 6.8 times larger. This happens because the lists of the formulas in monitors are growing for all the three properties. The largest trace that we could analyze without running out of memory for the property `lex_dup` is about 2100 messages, and for the properties `flight_info_dup` and `flight_seq_order` is about 2600 messages.

In Figure 8, we show how the total, verification, and event handling times change as the size of the trace increases for the property `flight_seq_order`. It can be seen that the time spent in monitors (verification time) is very close to the total time, and it grows exponentially with the size of the traces.

In a different experiment, we used the following property, which is a modification of the property `flight_seq_order`.

```
property('single-flight_seq_order) {
 always {
   case Dequeue(FlightPos(_, "TCF5934", _, _, _, _, date1)) =>
    state {
     case Dequeue(FlightPos(_, "TCF5934", _, _, _, _, date2)) =>
            date2.isAfter(date1)
    }
  }
}
```

By using a concrete value for the call sign, we only monitor the messages of the aircraft with the call sign "TCF5934". For this property, the monitor always maintains one formula at a time. We applied MESA to verify a message

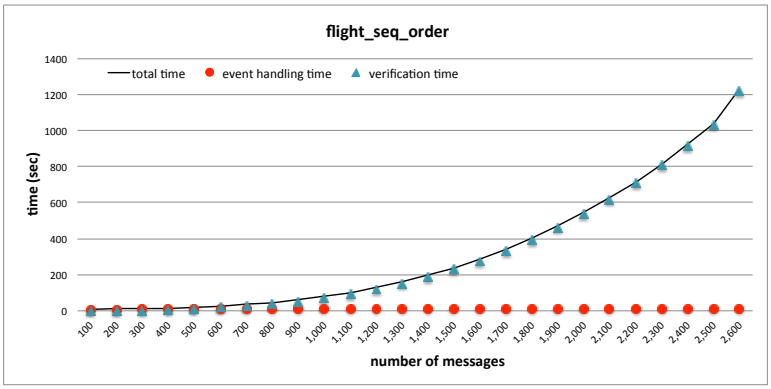

Fig. 8. Time for verifying the `flight-seq-order` property against SFDPS message sequences.

sequence of size 100,000, and 11 violations were detected. In Figure 9, we show how the total, verification, and event handling times change as the size of the trace increases for the property `single-flight_seq_order`. Unlike the property `flight_seq_order`, it can be seen that for this property, the time spent in monitors is very small, and the time spent on handling the events is very close to the total time. It can be also seen that the total time grows linearly with the size of the traces.

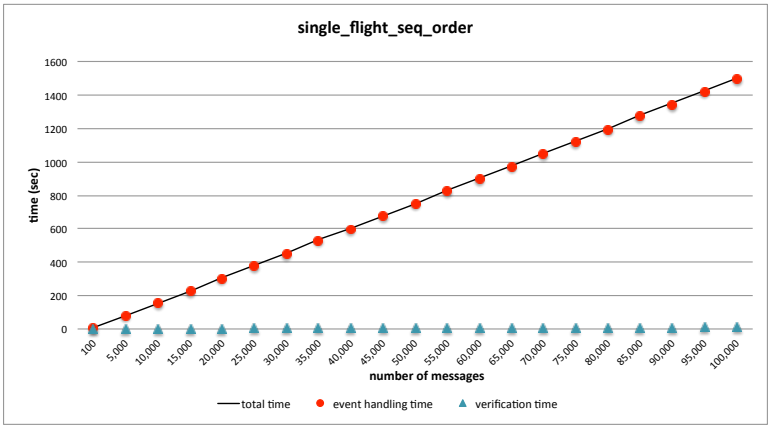

Fig. 9. Time for verifying the `single-flight-seq-order` property against SFDPS message sequences.

7 Discussion

In this section, we discuss the observations learned from our experiments, and also the current limitations of our approach.

As mentioned before, our experiments presented in Section 6 were performed offline. Our preliminary experiments show that MESA has similar performance for both offline and online message processing. However, in both cases MESA can run out of memory due to a large number of formulas to be monitored. We also show that monitoring the flight data for only one aircraft can considerably improve the performance, and eliminate the running-out-memory problem. However, the incoming live messages from the SFDPS service can come from more than 4,500 aircrafts, which may require maintaining more than 4,500 formulas in the monitor. Our evaluation suggests that a limitation of the approach is scalability for certain properties. One way to address this issue is to provide multiple instances of monitors running in parallel. For example, in the case of SFDPS message sequences, having simultaneously running monitor instances where each monitor only monitors data coming from certain aircrafts. To accomplish this, as part of our future work, we are looking to turn monitors into actor instances.

Prior to MESA implementation, one of the authors implemented several RACE actors that check certain properties such as looking for duplicate or out-of-order messages. Our preliminary experiments showed that checking requirements using such actors is faster than running the equivalent monitors in MESA. However, writing checks as actors is more involved than specifying properties at a high level, using LTL or state machines. The ease of writing properties seems to come at the performance cost. For example, consider the property `lex-dup`, formalized in 5 lines of code in a Scala DSL (see Section 3.2). A similar check implemented in RACE consists of 95 lines of Scala code.

In this work, we concentrated on verifying SFDPS feeds with respect to properties that should hold true for messages produced by SWIM. However, in the future, we would like to verify cross-component properties about messages that are both consumed and produced by SWIM. This would require applying MESA in a distributed setting, where message sequences to be monitored are coming from more than one source. We have already identified such properties about SWIM, and we would like to provide runtime verification for such properties.

8 Related Work

Runtime verification is a mature field of research with a large body of work and tools available. There are many flavors of approaches and tools, which can be categorized based on the domain of applications being monitored, instrumentation-based vs. nonintrusive approaches, online vs. offline monitoring, expressiveness of writing specifications, when the verdicts are returned (violation vs. validation), efficiency vs. usability, whether corrective action is taken, and many more. In this section, we mention some approaches related to our work.

In Aspect Oriented Programming (AOP), certain features are referred to as cross-cutting concerns because they cut across multiple modules and are orthogonal to the main functionality of the program. Examples of such features include logging, security management, and gathering metrics. Code instrumentation can also be implemented as aspects that observe the execution of the program and update the state of the monitors. All AOP frameworks have support for defining cross-cutting concerns (e.g., monitor update) as well as a way to specify when they are applied (e.g., when a specific method is invoked). ASPECTJ and JBoss are popular implementations of AOP for Java. Building on AOP approaches, there are domain-specific extensions, e.g., Tracematches [8] and J-LO [11]. Using these frameworks requires a considerable amount of code instrumentation to define aspects and their application. However, using TraceContract DSL gives users a more flexible and powerful interface to specify requirements.

Monitoring Oriented Programming framework MOP [4, 14] allows one to specify properties in several logical formalisms (LTL, finte state machines, extended regular expressions, etc.), which then get compiled into ASPECTJ aspects. Aspects can be embedded into the monitored system and the user can provide code to handle property violations or validations. MOP has different instances: JavaMOP for Java programs and BusMOP (for monitoring PCI bus traffic). MOP tools tightly integrate the generated monitors with the code and have the capability to change the program execution depending on the monitor handling code. However, our current goals do not include changing or repairing of the SWIM feed. We want to detect errors, report them and have SWIM errors fixed at the source of the problems.

9 Conclusion

In this paper, we presented MESA, a Message-Based System Analysis framework for runtime verification of safety-critical message intensive systems like SWIM used at NASA. MESA is built on top of RACE and TraceContract to address the main challenges of SWIM verification: RACE addresses connectivity and scalability issues and TraceContract enables nonintrusive trace analysis and property specification using a powerful internal DSL. We applied MESA to the SWIM flight data feed and caught duplicate and out-of-order messages.

As part of our future work, we would like to provide a functionality that allows for running monitors in a distributed setting to check for cross-component properties. We would also like to turn monitors into actors, where each actor monitors a property. In general, we plan to make MESA more efficient in terms of time and memory by utilizing functionalities that RACE provides in terms of data acquisition and data processing.

Acknowledgments

Authors would like to thank Klaus Havelund for help with TraceContract and specifying properties in TraceContract DSL. In addition, we would like to thank

Misty Davies and Klaus Havelund for helpful comments on the draft of this paper.

References

1. Akka - scalable realtime transaction processing, http://doc.akka.io/docs/akka/current/scala.html
2. FlightAware, https://flightaware.com
3. Lunar Atmosphere Dust Environment Explorer, https://www.nasa.gov/mission_pages/ladee/main/
4. MOP: Monitoring-Oriented Programming, http://fsl.cs.uiuc.edu/index.php/MOP
5. RACE: Runtime for Airspace Concept Evaluation , https://github.com/NASARace/race
6. Scala Programming Language, http://www.scala-lang.org/
7. FAA Telecommunications Infrastructure NEMS User Guide (2013)
8. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 345–364. OOPSLA '05, ACM, New York, NY, USA (2005)
9. Barringer, H., Havelund, K.: Tracecontract: A Scala DSL for trace analysis. In: FM: Formal Methods - 17th International Symposium on Formal Methods. pp. 57–72 (2011)
10. Barringer, H., Havelund, K., Kurklu, E., Morris, R.: Checking flight rules with tracecontract: Application of a Scala DSL for trace analysis (2011)
11. Bodden, E.: J-lo, a tool for runtime-checking temporal assertions (2005)
12. Luckenbaugh, G., Landriau, S., Dehn, J., Rudolph, S.: Service oriented architecture for the next generation air transportation system. In: Integrated Communications, Navigation and Surveillance Conference, 2007. ICNS'07. pp. 1–9. IEEE (2007)
13. Mehlitz, P., Shafiei, N., Tkachuk, O., Davies, M.: Race: building airspace simulations faster and better with actors. In: DASC: Digital Avionics Systems Conference (2016)
14. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14(3), 249–289 (2012)
15. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Foundations of Computer Science. pp. 46–57 (1977)