Feature

High-Integrity Runtime Verification

Alwyn E. Goodloe, *NASA Langley Research Center, Hampton, Va. USA*

Klaus Havelund, *Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA. USA*

Abstract—High-integrity systems must be very reliable, but their increasing complexity makes them difficult to test or formally verify. We introduce the concept of runtime verification that can be a critical element of an assurance case by guaranteeing that specifications hold at runtime. We show how runtime verification can be a trustworthy approach to assuring that critical systems are safe and dependable.

Introduction

The failure of high-integrity computing systems in application domains such as automobiles, aircraft, chemical plants, financial, and robotic systems can result in the loss of life, significant property damage, or large monetary loss. Recent incidents involving autonomous vehicles highlight the danger such systems pose to the public. Hence their correctness has become critical to maintaining a smoothly running society. High-integrity systems are often subject to regulatory oversight, where great emphasis is placed on assuring that the systems are safe, secure, and correct.

Within the field of software development, it is common to refer to the term "Verification & Validation" (V&V). Validation usually refers to ensuring that "the system does the right thing", and is typically a manual process involving inspecting the requirements and the behavior of a program to confirm they are what was intended. Verification, on the other hand, consists of ensuring that "the system does the thing right", and involves checking that the execution of the program satisfies certain assertions under all or many circumstances. Checking that it satisfies the assertions under *all* circumstances is usually performed using *formal verification* techniques, such as theorem proving and model checking. Checking that it satisfies the assertions under *some* (a finite set of) circumstances is referred to as *testing*. The activity of checking that *one* particular execution satisfies its specification is referred to as *Runtime Verification* [1], also commonly referred to with the acronym *RV*. RV can be applied to a system under observation (SUO) during test, but can, in particular, be applied after deployment of the system in the field during its entire operation. RV can be applied to log files produced by previous runs of a system, referred to as *offline* RV, or it can be applied during the execution of the system, referred to as *online* RV. Given that testing can fail to cover off-nominal conditions and formal proofs often rest on assumptions that may not hold in the field, RV should still be employed to ensure operational correctness. When a violation of the specification is detected, the RV system can either alert a responsible party, such as a drone safety pilot, or invoke an automated procedure that takes corrective action by *steering* the system into a safe state. Hence well engineered RV can provide guarantees against unsafe behavior.

We shall provide an introduction to RV in the context of high-integrity systems, drawing examples from case studies applying RV to aerospace systems. In the next section, we will present fundamental concepts and terminology. This is followed by sections dedicated to performing RV offline and performing RV online. We then address how to ensure that the RV system itself is trustworthy, a critical part of an assurance argument.

Fundamentals of Runtime Verification

XXXX-XXX © IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

Basic Concepts

Runtime verification is a dynamic software analysis technique that detects if a formally specified property is violated during an execution of a program or system. This can be applied offline by collecting a trace as a log file or online by monitoring the system whilst it is running. Runtime verification is therefore in some sense very limited in focusing on just one execution[1]. However, this focus in research over the last couple of decades has brought forward many interesting techniques, allowing for more interesting execution properties to be verified more efficiently and using more elegant specification languages.

In RV, the execution of a system is usually abstracted as an execution *trace* [2] of events or states. Say E is the set of events (or states) and E^* is the set of all finite sequences of elements of E, then a trace $\sigma \in E^*$ is a finite sequence of observed events or states. Traces in RV are always finite, in contrast to traces in formal verification, which are generally considered to be infinite in theory. In RV, a trace is checked against a formal specification $\phi \in \Phi$ in some specification logic Φ. Sometimes such a logic is referred to a *specification language*, or a *Domain-Specific Language* (DSL) for RV. A specification denotes the set of traces that satisfy it. That is, the *semantics* of the specification logic is modeled as a function $\mathcal{L}$ that maps a specification to a set of traces, formally $\mathcal{L} : \Phi \to \mathcal{P}(E^*)$. The RV problem then becomes that of checking that a trace σ is a member of the language of a specification ϕ, formally $\sigma \in \mathcal{L}(\phi)$. We also write this as $\sigma \models \phi$, stating that σ satisfies the specification ϕ.

Traces are extracted from the executing program. Assuming that $Prog$ is the set of all programs, the execution function is denoted by $Exec : Prog \times Input \to Output$, and the execution of a particular program P on input i by $Exec(P, i)$. Note that in the case of concurrent/interactive programs the situation is more complex. Of course one can attempt to monitor the output produced by the program, in which case we have our trace for free. Formally $Exec(P, i) \models \phi$. However, it is more common to instrument the program to produce a trace specifically for monitoring. Instrumentation is often performed manually by the programmer inserting logging statements in the code, but can also be automated. Automated instrumentation consists of modifying the original program with an instrumentation function $Instr : Prog \to Prog$, to produce a new program, which when executed delivers the execution trace to be analyzed. This way the verification problem can for formulated as:

$$Exec(Instr(P), i) \models \phi$$

In the following, we shall elaborate more on traces, specification logics, and code instrumentation. As an example, we shall consider monitoring something as simple as opening, reading, writing and closing of files.

Traces

There is no commonly agreed upon definition of what a trace is in the RV literature, neither with respect to what it should contain, nor what the format should be. With respect to contents, it completely depends on the application and the kind of properties one wants to monitor. It is, however, common for events to have a name indicating what kind of event it is, e.g. opening a file, reading a file, or closing a file. An example of a log consisting of three events (reporting opening, reading, and closing a file) is shown in Figure 1, in Comma Separated Value (CSV) format[2], JavaScript Object Notation (JSON) format[3], and Extensible Markup Language (XML) format[4] respectively. However, even within one format, one can represent a log in many different ways; we have just shown a few.

Note that in online monitoring, events are generated on the fly, and may appear as special messages in text format or even as data structures in a programming language. An example of this would be if a monitor is a process running in parallel with the monitored program, and events are messages sent from the monitored program to the monitor. In the case where a monitor is tightly inlined in the monitored program, it may even be able to see the internal state of the program, yielding yet a very different notion of trace.

[1] As will be discussed later, runtime verification can also be focused on multiple executions.

[2] CSV format: https://datatracker.ietf.org/doc/html/rfc4180

[3] JSON format: https://datatracker.ietf.org/doc/html/rfc7159

[4] XML format: https://www.w3.org/TR/REC-xml

CSV:

```
open,log1,rd
read,log1,1024
close,log1
```

JSON:

```
[
  {"event" :"open", "file": "log1", "mode": "rd"},
  {"event" :"read", "file": "log1", "bytes": 1024},
  {"event" :"close", "file": "log1"},
]
```

XML:

```
<log>
   <event>
     <name>open</name><file>log1</file><mode>rd</mode>
   </event>
   <event>
     <name>read</name><file>log1</file><bytes>1024</bytes>
   </event>
   <event>
     <name>close</name><file>log1</file>
   </event>
</log>
```

FIGURE 1. A log represented in CSV format, JSON format, and XML format.

Logics

As previously mentioned, a specification ϕ of a required trace property is expressed in a property specification logic Φ, formally $\phi \in \Phi$. Numerous different formalisms have been used for runtime verification. Some of the most commonly used include state machines, regular expressions and temporal logics. Such high-level property specifications are translated to monitors in a programming language, which then performs the actual trace analysis. A rich literature exists outlining different logics and how they are translated to monitors, or even directly executed as monitors.

Consider the file example. Suppose we want to specify the property that for any file f, if opened in read mode, it can only be read from, and if opened in write mode, it can only be written to, and an opened file should eventually be closed again. This property is formalized as a state machine in Figure 2.

Note that this state machine actually also says that after a file is opened it cannot be re-opened before it is closed again, and after that it cannot be closed again before opened again. This can also be expressed as a regular expression, which is a different way of expressing trace properties:

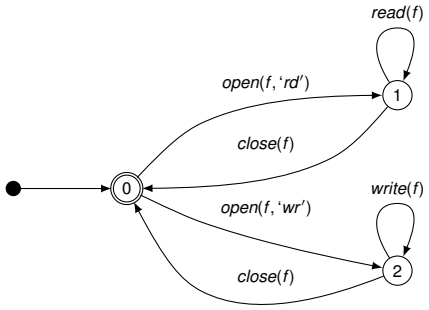

FIGURE 2. A state machine for the property: "*for any file f, if opened in read-mode, it should only be read from, and if opened in write mode, it should only be written to, and an opened file should eventually be closed again*".

$$(open(f, `rd')read(f)^* close(f))^* \mid$$
$$(open(f, `wr')write(f)^* close(f))^*$$

In many RV systems, properties are alternatively expressed as formulas in temporal logic. The following properties P_1 and P_2, for example, state that P_1: always ($\Box$), if a file is opened in some mode, it should eventually ($\Diamond$) be closed, and P_2: always, if a file is read, then in the past it was opened in read mode, and since (S) then it has not been closed.

$$P_1 : \forall f \in File, \forall m \in Mode.$$
$$\Box(open(f, m) \rightarrow \Diamond close(f))$$

$$P_2 : \forall f \in File \in Mode.$$
$$\Box(read(f) \rightarrow (\neg close(f) \ S \ open(f, rd)))$$

Property P_1 is also called a future time property since the $\Diamond$ operator refers to the future, while the property P_2 is called a past time property since the S operator refers to the past.

Instrumentation

We mentioned earlier the need for instrumenting programs to emit events to a monitor, and that this instrumentation can be manual or automated. A common approach to automated instrumentation is to use aspect-oriented programming. For example, AspectJ [3] is an aspect-oriented framework for Java. The AspectJ compiler takes as input a Java program and one or more aspects, and produces a new program, which is the original program plus

code weaved in as directed by the aspect. Figure 3 shows an aspect for inserting appropriate calls to a monitor of a class Monitor (not shown here), right before calls of functions for opening, reading, writing, and closing files. The Monitor class can be manually constructed to check our property, or it can be generated automatically from a specification.

```
public aspect MonitorAspect {
  pointcut open(int mode) : call(File.new(String,int)) &&
                            args(..,mode);
  pointcut read(File file) : call(String File.read(..)) && target(file);
  pointcut write(File file): call(void File.write(..)) && target(file);
  pointcut close(File file): call(void File.close()) && target(file);

  Monitor monitor = new Monitor();

  after(int mode) returning (File file) : open(mode) {
    monitor.open(file,mode);
  }

  before(File file) : read(file) {
    monitor.read(file);
  }

  before(File file) : write(file) {
    monitor.write(file);
  }

  before(File file) : close(file) {
    monitor.close(file);
  }
}
```

FIGURE 3. AspectJ aspect performing code instrumentation.

RV Frameworks

Runtime verification frameworks transform a formal specification written in a suitable logic into an executable monitor. RV frameworks can also generate supporting code to instrument the SUO in order to capture the trace of the execution. Depending on the framework, monitors can be implemented in a range of executable languages such as Python, Java, C, assembly language, or even as a Verilog program. Examples of RV frameworks include [4]–[14].

Interestingly, static analysis tools can potentially be used for (at least) offline monitoring. For example, the Cobra static analyzer [15] was used for log analysis as described in [16], where it is compared with an RV tool.

A Broader Definition of RV

Runtime verification as presented is defined as the activity of checking that an execution trace satisfies a property. However, the term can be perceived more broadly. In the broadest sense of the term, it represents the slogan: *"get the most out of your runs"*. The field, for example, also includes computing data beyond Boolean true/false verdicts from traces, and therefore overlaps with the field of data analysis, including production of statistics and visualization. Other aspects of RV includes machine learning, e.g. learning a specification of nominal behavior from a set of traces. This specification can then be turned into a monitor, checking that subsequent provided traces satisfy the specification learned from previous traces. Some systems allow formulating properties on a set of traces [17], also referred to as hyper properties (e.g. every pair of separate executions of a system must agree on the position of occurrences of some event e).

Offline Monitoring

In offline monitoring, the objective is to analyze a log file produced by a previous run of the SUO. Since the analysis is performed after the run, there is usually less emphasis on the monitor being high performance with respect to execution time, compared to the online case. Of course execution time does become a critical factor for very large log files with millions of events. A characteristic of offline monitoring is that there are less constraints concerning memory use, e.g. using dynamic memory allocation in a garbage collected language is commonly seen. These more relaxed constraints on the monitoring framework opens up for more expressive monitoring languages.

A classical distinction amongst Domain-Specific Languages (DSLs) for RV is that of external versus internal DSLs. An external DSL is a "small language" with its own grammar and parser. An internal DSL (sometimes referred to as an embedded DSL) is a library in a general purpose programming host language, such as Python, Java, or even C++ and C. An advantage of an internal DSL is that it offers the entire host language as part of the "logic", making it very expressive (Turing complete). When processing log files, it is often necessary to perform computations on the data in the log, such as extracting sub strings from strings in the log with regular expressions, computing averages, storing data points for later visualization, etc. For such purposes an internal DSL becomes very useful and even necessary.

An example of an internal DSL is PyCon-

tract[5] [18], a Python library offering a way of writing temporal properties using a combination of state machines and rule-based programming. States can be parameterized with data, effectively representing *facts*, which are stored in the monitor memory, as is typical for rule-based systems. As an example, we shall illustrate how to program a monitor in PyContract for the following property about the execution of commands on board a spacecaft. A command, identified by a name, is first dispatched, after which it must complete execution within 3 seconds (3000 milliseconds), without failing before then. Furthermore we would like to be informed of the average execution time for each command, computed over all executions of the command.

Figure 4 shows a monitor for the command execution property. The monitor is defined as a class (line 3) instantiating a statistics class (line 4) which we shall not elaborate on. It just makes the point that one can create any statistics desired as part of a monitor. The transition function (lines 6-9) is always enabled. Whenever an event is submitted to the monitor, this function is applied to the event. In this case, if the event is a dictionary with a `name` field having the value `dispatch`, a `cmd` field and a `time` field, then the monitor enters a `DoComptete(c, t)` state, parameterized with the command `c` and time `t`. The `DoComplete` state itself has a transition function, which looks out for three types of events: the failure of the command, a timeout of more than 3 seconds since the dispatch, or a successful completion of the command.

The monitor can be instantiated and fed a trace to verify as shown in Figure 5. Here a trace, for illustration purposes, consisting only of seven events, is constructed directly as a Python list of dictionaries (each dictionary representing an event). In practice, a trace is typically read in from a file. The PyContract API also supports feeding the monitor with events one by one, instead of providing a trace. It can therefore technically also be used for online monitoring. The trace in Figure 5 violates the specification twice: the TURN command completes after 3 seconds, and the THRUST command never completes. The SEND command completes twice with an average execution time of 750 milliseconds. The output is shown in Figure 6.

[5] https://github.com/pyrv/pycontract

```
1  import pycontract as pc
2
3  class Commands(pc.Monitor):
4      statistics = Statistics()
5
6      def transition(self, event):
7          match event:
8              case {'name': 'dispatch', 'cmd': c, 'time': t}:
9                  return Commands.DoComplete(c, t)
10
11     @pc.data
12     class DoComplete(pc.HotState):
13         cmd: str
14         time: int
15
16         def transition(self, event):
17             match event:
18                 case {'name': 'fail', 'cmd': self.cmd}:
19                     return pc.error()
20                 case {'time': t} if t - self.time > 3000:
21                     return pc.error()
22                 case {'name': 'complete', 'cmd': self.cmd, 'time': t}:
23                     self.monitor.statistics.add(self.cmd, t - self.time)
24                     return pc.ok
```

FIGURE 4. Defining a monitor using the PyContract library.

```
1  m = Commands()
2  trace = [
3    {'name': 'dispatch', 'cmd': 'TURN', 'time': 1000},
4    {'name': 'dispatch', 'cmd': 'THRUST', 'time': 4000},
5    {'name': 'complete', 'cmd': 'TURN', 'time': 6000},
6    {'name': 'dispatch', 'cmd': 'SEND', 'time': 6000},
7    {'name': 'complete', 'cmd': 'SEND', 'time': 7000},
8    {'name': 'dispatch', 'cmd': 'SEND', 'time': 7500},
9    {'name': 'complete', 'cmd': 'SEND', 'time': 8000},
10 ]
11 m.verify(trace)
12 m.statistics.show()
```

FIGURE 5. Applying the monitor defined in Figure 4.

```
1  *** error transition in Commands:
2      state DoComplete('TURN', 1000)
3      event 3 {'name': 'complete', 'cmd': 'TURN', 'time': 6000}
4
5  *** error at end in Commands:
6      terminates in hot state DoComplete('THRUST', 4000)
7
8  Average command durations:
9  SEND: 750.00
```

FIGURE 6. Output from monitor application in Figure 5.

An application of PyContract to the analysis of real telemetry from NASA's Europa Clipper mission is described in [19]. An application of the internal DSL TraceContract to verify command sequences before being sent to the Lunar spacecraft LADEE is described in [20]. TraceContract is a library in the Scala programming language, and is similar

to PyContract. Although this is a form of program verification, it has similarities with log analysis.

Online Monitoring

The objective of online monitoring is to verify that the SUO adheres to a formal specification during execution, meaning an execution trace must be collected and analyzed while the system is running, possibly requiring an action that steers the system to a safe state if the specification is violated. Monitoring languages and frameworks targeting online monitoring are often customized for particular domains such as network infrastructure, cloud computing, and cyber-physical systems (CPS). In this section, we will focus on RV for CPS.

Safety-critical cyber-physical systems are usually subject to stringent restrictions such as hard real-time deadlines, size, weight, and power, thus online monitors of such systems must also conform to the same restrictions. Resource limitations can restrict online RV to monitors that run in constant time and space, and limits the size of the trace that can be kept in memory. Extensive instrumentation of the SUO, allowing internal variables to be observable, may complicate real-time scheduling and certification. Hence, a rule of thumb often followed in online monitoring is to go with light-weight instrumentation when possible. Moreover, the monitor must detect the specification violation in a timely manner, with enough time for the corrective action to prevent catastrophic failure.

NASA's Copilot[6] [7] is representative of RV frameworks targeting online monitoring of CPS systems. Copilot's stream based specification language is an internal DSL embedded in the programming language Haskell. The framework generates semantically equivalent C99 monitors that execute in constant space and constant time. Safety properties of CPS systems are usually expressed in terms of continuous physical values such as GPS coordinates or airspeed; hence, Copilot and other similar RV frameworks obtain a trace by sampling the sensor data and system state at suitable rates.

Consider a fixed-wing unmanned aircraft controlled by an AI enabled autopilot, where RV is being used to ensure safe operation of the aircraft. A stall is an aerodynamic condition where the aircraft angle of attack of a wing is greater than the designated critical value, causing the wing to cease generating lift. To prevent the autopilot from accidentally putting the aircraft into a stall, we can write a monitor specification expressing that if the aircraft maintains an angle of attack greater than the designated stall angle (14°) for 40 seconds, then invoke a function, `move_to_safe_AA` (code not shown), that overrides the autopilot putting the aircraft into a safe state.

Figure 7 shows a Copilot module `DetectStall` specifying this property. A Haskell extension is used. The first three lines of the module import the necessary packages. The declarations `angleOfAttack` and `sysClk` define streams obtained by sampling system sensor data measuring the angle of attack (`angle_attack`) as well as a stream of integral clock values, (`sys_clock`), sampled at same rate as the sensor data. Copilot's past-time metric temporal logic `alwaysBeen` operator is used to specify the property. If the property is true, the trigger invokes the function to lower the angle of attack. The last line of the definition synthesizes a C monitor from the high-level definition. There is a straightforward translation from each Copilot stream value to a C value, where compound stream values such as structs and arrays are translated to C structs and arrays with corresponding struct field names and array lengths. While Copilot streams are conceptually infinite sequences, Copilot-generated C programs only use a finite amount of memory, with each stream translated to a ring buffer. Copilot monitors are scheduled as real-time tasks so that data and events are sampled at the rate needed to produce the desired trace.

In over a decade of use in NASA unmanned aircraft flight tests, Copilot has been a foundational element of fault-tolerant monitors [21] and has been used to monitor properties such as aircraft stall, battery life, safe air traffic separation, safe flight mode transitions, and flight plan violation detection. Copilot has been classified as a software engineering tool by NASA so that the monitors generated by the framework can be certified for use on mission critical systems.

Trustworthy RV

News reports are awash with stories of misplaced trust in new technology resulting in unpleasant consequences. If RV is to play a central role in assuring

[6] https://github.com/Copilot-Language/copilot

```
{−# LANGUAGE RebindableSyntax #−}

module DetectStall where

import Language.Copilot
import Copilot.Compile.C99
import qualified Copilot.Library.MTL as
                    Q hiding (trigger)

sysClk :: Stream Word64
sysClk = extern "sys_clock" Nothing

angleOfAttack :: Stream Double
angleOfAttack = extern "angle_attack" Nothing

unsafeAngleOfAttack aattack clk' =
    Q.alwaysBeen 0 40 clk' 1 (aattack > 14)

spec = do
  trigger "move_to_safe_AA"
      (unsafeAngleOfAttack angleOfAttack sysClk) []

main = reify spec >>= compile "stall_detect"
```

FIGURE 7. Defining a Copilot monitor for aircraft stall.

high-integrity systems, then the RV must be demonstrated to be *trustworthy* [22]. To ensure that our trust in RV is not misplaced, RV for high-integrity systems must at least:

- Support critical certification requirements.
- Tolerate faults and security threats.
- Support validation that the specification of the monitor is correct.
- Provide evidence that the synthesized monitor does indeed implement the specification.

High-integrity systems are often subject to certification and regulatory requirements such as DO-178C [23] in civil aviation. RV being used on such a system will also have to satisfy the regulations. Let us look at requirements commonly found across a range of certification guidelines. To ensure that the requirements and safety analyses performed early in systems development are reflected throughout the lifecycle, regulations often require documentation of traceability from requirements to object code. To satisfy this requirement, monitor generation frameworks should produce documentation that supports traceability from specification to monitor code. In addition, verification must be performed to demonstrate that the monitors do indeed implement their specification. Although extensive testing is the conventional means of verification, advances in automated formal verification have been successfully applied to the task [24], providing mathematical proof that the monitor and specification are equivalent. Another common guideline that RV monitors need to respect is for software to have error handling for out-of-bounds data. Thus it is essential for sound software engineering practices to be applied to RV.

Writing formal specifications is an error prone task and getting a RV specification wrong can have disastrous consequences. However, the RV framework can provide features to assist the user in getting the specification right. Visualizing logical specifications [25] can illuminate errors in complex specifications. A feature allowing the direct execution of the specification on sample data while eliding issues involving integration with the SUO can also aid in validation. Many CPS systems are designed using modern model-based engineering tools such as MATLAB Simulink [7] that have a significant simulation capability. Including interfaces to such simulation tools can assist in validating specifications. There are cases where the best means of validating a specification is to prove theorems about it. Incorporating the capability to prove theorems about a specification directly into an RV framework makes this much easier.

The design process for high-integrity systems typically includes a hazard analysis [26] to identify hazards that can lead to catastrophic failure and a threat analysis to identify security threats. When integrated with an SUO, RV monitors need to be able to tolerate the faults identified in the safety analysis. Given that fault-tolerance is typically achieved by architectural means, the monitors may have to run on redundant hardware and employ voting to achieve the desired level of fault tolerance or be run in a separate partition to achieve an acceptable level of independence. Many security threats can be mitigated by following secure coding practices. However, one can envision situations where monitors and associated steering actions are exploited in denial-of-service attacks or are used to prevent a vehicle from reaching its destination.

[7] MATLAB and Simulink are registered trademarks of The MathWorks, Inc.

Conclusion

We have presented an overview of runtime verification, which has been the subject of active research for over two decades. The technology is now sufficiently mature that it is being transitioned into practice, as is illustrated by examples of both offline and online monitoring using RV frameworks developed in part at NASA. Offline RV can lower cost and complexity by making log analysis easier, more systematic, and repeatable from project to project, whereas online RV enables the assurance of systems that might otherwise be too complex to do so via conventional means. However, it is not an end-run around assurance as the monitors themselves must be verified entailing some cost. Making RV trustworthy enough to be a cornerstone of a certification argument will be a considerable engineering challenge that the community will have to meet for RV to fulfill its potential.

Acknowledgements

The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by Alwyn Goodloe was supported by the System-Wide Safety project in NASA's Airspace Operations and Safety Program.

REFERENCES

1. Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification," in *Engineering Dependable Software Systems*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Broy, D. A. Peled, and G. Kalus, Eds. IOS Press, 2013, vol. 34, pp. 141–175.
2. G. Reger and K. Havelund, "What is a trace? a runtime verification perspective," in *ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, ser. Lecture Notes in Computer Science, 2016, pp. 339–355, 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016) ; Conference date: 05-10-2016 Through 14-10-2016.
3. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, ser. LNCS, J. L. Knudsen, Ed., vol. 2072. Springer, 2001, pp. 327–354.
4. C. Colombo, G. J. Pace, and G. Schneider, "LARVA — safer monitoring of real-time Java programs (tool paper)," in *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 33–37. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2009.13
5. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Techniques for Technology Transfer*, pp. 249–289, 2011, http://dx.doi.org/10.1007/s10009-011-0198-6.
6. S. Hallé and R. Villemaire, "Runtime enforcement of web service message contracts with data," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 192–206, 2012.
7. I. Perez, F. Dedden, and A. Goodloe, "Copilot 3," NASA Langley Research Center, Tech. Rep., 2020.
8. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu, "Monitoring of temporal first-order properties with aggregations," *FMSD*, vol. 46, no. 3, pp. 262–285, 2015.
9. G. Reger, H. C. Cruz, and D. Rydeheard, "MarQ: Monitoring at runtime with QEA," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, C. Baier and C. Tinelli, Eds. Springer, 2015, pp. 596–610. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_55
10. N. Decker, M. Leucker, and D. Thoma, "Monitoring modulo theories," *Software Tools for Technology Transfer (STTT)*, vol. 18, no. 2, pp. 205–225, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10009-015-0380-3
11. H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, and A. Weiss, "TeSSLa – an ecosystem for runtime verification," in *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, ser. LNCS, T. Dang and V. Stolz, Eds., vol. 13498. Springer, 2022, pp. 314–324.
12. F. Gorostiaga and C. Sánchez, "HStriver: A very functional extensible tool for the runtime verification of real-time event streams," in *Formal Methods*, ser. LNCS, M. Huisman, C. Păsăreanu, and N. Zhan, Eds.,

vol. 13047. Springer, 2021, pp. 563–580.
13. K. Havelund, “Data automata in Scala,” in *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*. IEEE Computer Society, 2014, pp. 1–9.
14. F. Chen and G. Roşu, “MOP: an efficient and generic runtime verification framework,” in *Object Oriented Programming, Systems, Languages, and Applications*, 2007, pp. 569–588.
15. G. J. Holzmann, “Cobra: a light-weight tool for static and dynamic program analysis,” *Innov. Syst. Softw. Eng.*, vol. 13, no. 1, pp. 35–49, 2017. [Online]. Available: http://spinroot.com/cobra/
16. K. Havelund and G. Holzmann, “Programming event monitors,” *International Journal on Software Tools for Technology Transfer*, 2023.
17. S. Stucki, C. Sánchez, G. Schneider, and B. Bonakdarpour, “Gray-box monitoring of hyperproperties,” in *International Symposium on Formal Methods: Formal Methods – The Next 30 Years*, ser. LNCS, vol. 11800, September 2019, pp. 406—-424.
18. D. Dams, K. Havelund, and S. Kauffman, “A Python library for trace analysis,” in *22nd International Conference on Runtime Verification (RV)*, ser. LNCS, T. Dang and V. Stolz, Eds., vol. 13498. Springer International Publishing, 2022, p. 264–273.
19. B. Duckett, K. Havelund, and L. Stewart, “Space telemetry analysis with PyContract,” in *Applicable Formal Methods for Safe Industrial Products - Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday*, ser. LNCS, A. E. Haxthausen, W. Huang, and M. Roggenbach, Eds., vol. 14165. Springer International Publishing, 2023.
20. E. Kurklu and K. Havelund, “A flight rule checker for the LADEE Lunar spacecraft,” in *17th International Colloquium on Theoretical Aspects of Computing (ICTAC’20)*, ser. lncs, vol. 12545, 2020.
21. L. Pike, N. Wegmann, S. Niller, and A. Goodloe, “Copilot: monitoring embedded systems,” *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 235–255, 2013.
22. A. Goodloe, “Challenges in high-assurance runtime verification,” in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, 2016, pp. 446–460.
23. RTCA, “Software considerations in airborne systems and equipment certification,” RTCA, Inc., 2011, RCTA/DO-178C.
24. R. G. Scott, M. Dodds, I. Perez, A. E. Goodloe, and R. Dockins, “Trustworthy runtime verification via bisimulation (experience report),” in *Proceedings of the 28th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2023.
25. J. Scott-Brown and A. Papachristodoulou, “Visualization of temporal logic specifications,” in *Proceedings of the Eurographics/IEEE VGTC Conference on Visualization: Posters*, ser. EuroVis 2017. Eurographics Association, 2017, p. 117–119.
26. SAE International, “Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment,” SAE International, 1996, aRP 4761.

Alwyn E.Goodloe is a research computer engineer at the NASA Langley Research Center where his research focuses on developing formal methods and tools targeting the verification of safety-critical systems. He received his Ph.D. in Computer and Information Science at the University of Pennsylvania. He is a member of the IEEE and ACM. Contact at a.goodloe@nasa.gov.

Klaus Havelund is a senior research scientist at the NASA Jet Propulsion Laboratory where his research focuses on developing and applying formal methods and tools, and in particular such based on runtime verification. He received his Ph.D. in Computer Science at the University of Copenhagen, in part carried out at École Normale Supérieure in Paris, France, and University of Aalborg, Denmark. He is a member of ACM. Contact at klaus.havelund@jpl.nasa.gov.