What is a Trace? A Runtime Verification Perspective

DOI:
10.1007/978-3-319-47169-3_25

Document Version
Accepted author manuscript

What is a Trace? A Runtime Verification Perspective

Giles Reger[1]* and Klaus Havelund[2]**

[1] University of Manchester, Manchester, UK
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. Runtime Monitoring or Verification deals with *traces*. In its most simple form a monitoring system takes a trace produced by a system and a specification of correct behaviour and checks if the trace conforms to the specification. More complex applications may introduce notions of feedback and reaction. The notion that unifies the field is that we can abstract the runtime behaviour of a system by an execution trace and check this for conformance. However, there is little uniform understanding of what a trace is. This is most keenly seen when comparing theoretical and practical work. This paper surveys the different notions of trace and reflects on the related issues.

1 Introduction

Runtime Monitoring or Verification [29, 45] is a form of dynamic analysis where a system of interest is abstracted as an execution trace. The most common notion of runtime verification is to take a specification of correct behaviour ϕ and a trace τ and check for language inclusion i.e. $\tau \stackrel{?}{\in} \mathcal{L}(\phi)$. This can be applied *offline* by collecting a trace as a log file or *online* by monitoring the system whilst it is running.

How the trace τ is captured from the system and described for the monitoring process is important. We observe that currently (i) there is no general notion of what should be recorded in a trace, and (ii) there is no general format for recording traces as log files. This state of affairs hinders interoperability of runtime verification tools, sharing of case studies and benchmarks, and application of runtime verification techniques (as time must be spent deciding how to generate traces).

A solution would be to develop a general trace format to be used in runtime verification that is optimal from the perspective of runtime verification tools. However, we also observe that recording log files or execution traces is common in many areas of software engineering where runtime verification is not used. To encourage use of (formal) monitoring techniques in such areas it is important to understand the kinds of traces they deal with.

Therefore, we begin in Sections 2 and 3 by reviewing existing notions of trace, both within and outside the runtime verification community. Then we discuss two important

* The contribution of this author is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

** 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.

points. In Section 4 we discuss what should appear in a trace, and in Section 5 we discuss what format a trace file should take. Finally, Section 6 concludes with a discussion of what further issues to take into account when considering general trace formats.

2 Traces in Runtime Verification

We briefly review the role and occurrence of different notions of trace within the field of runtime verification.

2.1 Traces as Models

Traces are typically introduced as models of specifications written in a specification language. It is common to say that a specification φ denotes a (usually infinite) set of *traces*. More precisely one would normally define a signature Σ which induces a set of possible interpretations or traces $\mathcal{T}$, and define a denotation $|\varphi| \subseteq \mathcal{T}$, i.e the conditions for the trace to be a *model* for φ. As discussed below, the signature and the form of traces built from it can vary. But in general the signature will capture a notion of *event* and traces will be (finite) sequences of events.

Propositional Traces. The most simple case of this is when regular expressions or state machines are used as specifications and traces are taken as finite traces of propositional symbols. For example, the regular expression specification $(ab)^*$ has the trace $abab$ in its language but not the trace $abbab$. Another popular language for runtime verification is Linear Temporal Logic (LTL). In the propositional case it is common to have a one-event-at-a-time assumption. This makes the traces similar to those above. However, in contrast to model checking, traces are usually considered finite, and there have been various proposals for finite-trace versions of LTL. Relaxing the one-event-at-a-time assumption means that multiple events may occur at each time point, leading to a sequence of sets presentation. For example, assuming some appropriate finite-trace semantics, the LTL specification $\Box(a \rightarrow \Diamond(b \wedge c))$ has the trace $\{a\}.\{\}.\{b\}.\{b, c\}$ in its language. More complicated notions of trace tend to build on these ideas and we briefly cover some of these in the following.

Adding Time. The traces above introduce a *qualitative* notion of time i.e. give an ordering of events. However, they do not capture the *quantitative* distance (in time) between events. There have been various extensions of specification languages to deal with this. We can add clocks to automata to get timed automata [2, 17] and add intervals to get timed regular expressions [4]. Intervals can also be added to LTL to get extensions such as Metric Temporal Logic (MTL) [44, 54] and Timed Propositional Temporal Logic (TPTL) [3].

Such specifications often come with two alternative notions of trace or variations on their semantics. The first is to take a *pointwise semantics* as before where specifications denote (possibly infinite) *timed words* that add (strictly increasing) real-valued timestamps to events (or sets of events). For example, the MTL specification $\Box(a \rightarrow \Diamond_{(1,3)} b)$

has the trace $(1, a)(1.5, a)(3.5, b)$ in its language but not $(1, a)(1.5, a)(2.4, b)$ (again assuming an appropriate finite trace semantics). The second is to consider a *continuous semantics* where the trace is captured by a signal function $f : \mathbb{R}_+ \to 2^{\Sigma}$ mapping time t to a set of events $f(t)$ holding at time t. A signal function can be transformed into a timed word and a timed word can be transformed into a signal function if they satisfy properties that mean that an infinite number of events cannot occur in a finite amount of time. For signal functions this is called the *finite variability* property and for timed words this is called the *non-Zeno* property. Pragmatically, the difference between the pointwise and continuous view is that the former has an event-driven quality, and the second more intuitively reflects a setting where a system is periodically sampled.

Adding Data. Relatively early on in the history of runtime verification it was noted that it was useful to add a notion of data to specification languages, in order to capture traces with data carrying events. There is a wide range of different approaches to specification but the underlying traces are similar although the context of the specification language often leads to differing terminology when talking about traces.

There is a large body of work [1, 7–9, 20, 49] dealing with traces as (finite) sequences of so-called *parametric events* of the form $e(a, b, c)$, where e is an event name and a, b, c are data values. We note that not all of these work uses the term *parametric event*, but the overall concept remains the same. A separate effort is built on the existing theory of *data words* captured by *register automata* [26, 32]. The notion of trace is the same i.e. a trace is a sequence of letters from a finite alphabet paired with a data value from some infinite set. We mention this work separately as it is associated with a large body of theoretical work from outside of runtime verification [41, 52, 61].

A few extensions naturally lift propositional temporal logic to first-order, introducing functions, predicates, and quantification. As per the standard lifting, interpretations must be extended to interpret function and predicate symbols, thus leading to additional information being added to the trace as in [15, 25]. Although we note that it is unusual to do this in practice. It is common [15] for specifications to restrict functions and predicates to some well-defined *theory* such as arithmetic. In this case the interpretation of such symbols is implicitly defined by the theory. An approach to monitoring called *monitoring modulo theories* [25] (following satisfiability modulo theories) considers traces mixing theory symbols and so-called *observation* symbols capturing system events.

In cases where specification languages include a notion of *quantification*, the domain of quantification must be captured in some way. In some situations it would be reasonable for this to be captured in the specification (e.g. if quantifying over some fixed set of values) but more typically it is considered part of the trace. Whilst it would be reasonable to define this domain separately it is most common [7, 20, 58] for it to be defined exactly as values extracted from the trace. Independently of how the quantification domain is captured there is also the consideration of whether it should be fixed throughout the trace. For example, some approaches [68] quantify over values seen in the trace *so far* rather than the whole trace. Such decisions can significantly alter the interpretation of the specifications.

As a final note, once the notion of event is no longer propositional it is possible to introduce the notion of events with complex *structure*. So far, in our discussion, events

have had a flat form consisting of a name and a sequence of data values. This may not reflect real-world scenarios where data structures often encapsulate other nested structures, and recorded observations therefore may have structured fields. However, it is not common to consider structured events, although such efforts have been seen. For example, in [35] a first-order temporal logic is defined over XML documents where events are structured XML records.

Data and Time. One can add both data and time to produce a language that has more complex traces. One can, of course, treat time as just another element of data, and this is often sufficient. However, approaches that do this do not lend particular support for special metric operators as the specification language is not aware of the special status of this data.

One example of a language that combines data and time is Metric First Order Temporal Logic (MFOTL) [16] defined over traces consisting of timestamped parametric events. Another example is Signal Temporal Logic (STL) [48] where (continuous semantics) MTL is extended so that signals are real-valued (rather than boolean-valued). Implicitly this means that each propositional event has a real-valued parameter. Properties can then place conditions on functions of signals over time. There is also an extension of STL that deals with time-frequency analysis where traces are represented by a spectogram (a two-dimensional representation of time versus frequency) [27].

We would also like to mention a specific form of structured data in traces coming from the related field of statistical model checking. Spatial-Temporal logics [12, 31, 33, 34] are defined over sequences of spatial structures, for example quarternary tree structures [34]. This is an example of an application domain where the structure of data is well-defined and the specification language is interpreted over this structure.

2.2 Instrumentation Techniques

Another way of viewing what traces are, from a runtime verification perspective, is to consider the different instrumentation techniques commonly used in runtime verification as these reflect what is being observed.

Instrumenting Java. The most common approach in the literature for observing events in Java programs is to use AspectJ [6]. This approach tends to be used for intercepting method calls and their parameters. Therefore, event names typically relate to method calls with a few exceptions, for example taking and releasing locks. However, early work on Java instrumentation was more general. The Java-MaC [42] tool introduces a low-level language for identifying events in terms of program variables and fields in objects. Alternative approaches to Java instrumentation include JVM agents (used by RV-Monitor [47]) and Java Reflection (used by JUnitRV [24]). Again these techniques tend to focus on method invocations, but are not restricted to these.

Instrumenting C. Instrumentation techniques for the runtime verification of C (and C++) programs are less well established. There exists some work on extending the AOP

approach to C, but with relatively little uptake compared to AspectJ. RMOR [36] is a framework for monitoring the execution of C programs against state machines using an aspect-oriented pointcut language similar to AspectJ's. The system is implemented in the C analysis and transformation package CIL [22], which itself is programmed in Ocaml. AspectC++ [5] is a mature framework for aspect-oriented programming in C++. InterAspect [66] provides a GCC-plugin that supports pointcut definitions for the GIMPLE intermediate language. Finally, [21] reports on recent promising work and gives a good overview of previous efforts. The two main other approaches are manual instrumentation and code rewriting (as done by E-ACSL [43] and RiTHM [51]). For C programs it is more common to take changes in variable values as events. Another large consideration for C programs is that of memory safety, and this is a common behaviour to observe.

Other Languages. Whilst Java and C remain the two most common languages considered for monitoring, there exists some work on monitoring other languages. There is a recent body of work monitoring Erlang programs (ELARVA [23] and detectEr[3]). Initially, this work took advantage of Erlang's tracing mechanism to hook into Erlang's virtual machine to receive events as messages [23]. More recent work [18, 19] employs an Aspect-Oriented Programming framework to inject instrumentation as in AspectJ. It should be noted that when monitoring Erlang programs there is an additional issue of distributed (asynchronous) computation to contend with. There is also some work on monitoring Python programs [59] which employs function decorators to modify functions with additional instrumentation.

Hardware Instrumentation. For hardware monitoring there appears to be two main approaches [70]. The first is to add a passive device to the system bus and "sniff" ongoing activities. For example, BusMOP [56] uses this approach to detect I/O accesses, memory accesses and interrupts. The second approach is to directly access relevant signals and registers and compile the property to be monitored directly to a circuit [39, 46, 62, 69]. The former approach is event-triggered whilst the second samples a continuous signal on clock cycles.

3 Traces Elsewhere

In this section we briefly discuss sources of traces in areas that have received a mixed level of attention from the runtime verification. Understanding what traces are available here could be useful in understanding what kinds of traces we should be dealing with.

Web Servers. Web servers typically log accesses and errors. There are a number of standards for access logs supported by the main web server technologies. For example, the Common Log Format[4] logs each access as a single line consisting of the host, an

[3] `http://www.cs.um.edu.mt/svrg/Tools/detectEr`

[4] `https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format`

identity, the user, a date, the request, the status and the number of bytes. Within this the identity, timestamps and statuses are also standardised. Whilst this format is quite straightforward, and looks similar to what we discussed above, the more complex (draft) Extended Log Format[5] uses a header to specify the data types in each field.

Databases. Database systems typically log transactions to guarantee durability of data. As these logs are meant for internal consumption there is little available about the format of such logs. The information stored would typically involve the query type and any arguments. As these logs are used to ensure durability they are typically circular i.e. older records are overwritten by newer records once the changes in older records have been flushed to main storage.

System Logging in Unix. When exploring security related issues it is common to observe the system calls made to the Unix kernel. Tools for this task include `strace` and `ltrace`. `strace`[6] attaches to a process and records system calls and signals. Each line of the log file records the system call name followed by its arguments in parenthesis and its return value. For example,

```
open("/dev/null", O_RDONLY) = 3
```

records `open` being called with a pathname and flag. Errors include the relevant error number and string, for example

```
open("/foo/bar", O_RDONLY) = -1 ENOENT (No such file or directory)
```

Additionally, signals are printed as signal symbol and decoded siginfo structure. For example,

```
sigsuspend([] <unfinished ...>
--- SIGINT {si_signo=SIGINT,si_code=SI_USER,si_pid=...}
+++ killed by SIGINT +++
```

is an excerpt from stracing and interrupting the command "sleep 666". `ltrace` is very similar to `strace` but intercepts calls to dynamic libraries. This could be used to detect calls to the standard C library (for example).

System Logging in Windows. As expected, Windows systems have more propriety logging facilities than Unix-based systems. However, one can access *Windows Event Logs*, which record a range of different events occurring in the system. Recorded events contain a source, category, identifier and an event-specific string such as a filename or username. Events can be of a number of kinds, e.g. application events, system events, security events. There has been some work filtering and extracting well-defined events to perform log checking [60] but it seems that this is not well-supported.

[5] `https://www.w3.org/TR/WD-logfile.html`

[6] This information is based on that found in `man strace`.

4 What Should Go Into the Trace

Let us attempt to summarise the previous review to draw some conclusions about the kinds of things that should be supported in traces (ignoring the format of such things for now). Firstly, it is clear that we need good support for *Data* and *Time* as these are fundamental. In addition, we note that separate support for time is beneficial as it may have special requirements (e.g. is strictly increasing) not shared by normal data parameters. Beyond this there are a number of ideas to discuss.

Assumptions about relevance The main observation from Section 2.2 is that RV activities tend to specify the required events, record such events in a trace, and perform monitoring on that trace. However, many log files produced for other purposes will attempt to record as much relevant information as possible. Any useful system for recording and using traces would likely need to support traces containing more information than is relevant to the task in hand.

Do we need event names? It would seem that each event should be given a name. However, this could be an overly rigid requirement coming from a particular view of monitoring. In the setting of hardware monitoring where one has a number of signals one is observing, there is no notion of event name, as each event contains the same information i.e. a vector of values. Perhaps forcing an event to have a name is therefore restrictive.

Ordered or named parameters? Previously we assumed events had a fixed number of parameters and the values for each parameter was included in events. However, this does not allow for two scenarios. Firstly, where there are a large number of parameters and only a few are relevant. Secondly, where there may be a variable number of parameters but we know that the particular parameter of interest will be present. These cases can be addressed by identifying parameters by a *name* rather than a position. Supporting these alternative presentations would extend applicability to these scenarios.

Structured events. Most observations mentioned in the previous section are flat e.g. system calls have a flat list of parameters. However, there are some notable contexts where data values may have structure that should be recorded. A simple case is where the data value is a collection of other data values i.e. has variable non-fixed size. Two places where this issue is likely to occur is when serialising data structures and in Web Services, where structured data is common-place.

Notions of equality and other types. There is often an implicit direct notion of equality i.e. that one can compare data values in the trace directly. However, this may be too simplistic. For example, if one were to record memory addresses to identify data structures then equality should be interpreted within the context of memory management or garbage collection, which indicates the lifetime of that identity[7]. When richer

[7] The issue here is that over the lifetime of a program the same memory address could refer to different data structures if some structures are deleted.

data values appear in the trace there may be non-standard interpretations for equality operations on them. In either case, additional information may be required to interpret the trace.

The MetaData. One may need more information to understand the trace than is described in the events. For example, domains of quantification, the relevant signature/alphabet, sampling information, units for certain measurements etc. It would be useful for a trace format to support additional (structured) metadata of this kind. Otherwise a separate file will be required.

Capturing context. For some monitoring activities there is a wider context that may be relevant. For example, when monitoring Java applications it may be relevant when a certain (monitored) object is garbage collected. This could be encoded as events.

5 What Format Should a Trace File Take?

It seems clear that traces should follow a well-defined common format. This allows parsing and printing tools, as well as libraries to be easily reused. This was the approach taken by the RV competition [10, 11, 30], and we here review the file formats used for this competition, highlight their advantages and disadvantages, and point out some alternative uses of these formats.

Comma Separated Values (CSV). This standardised[8] format is often used for the storage and transport of simply structured data. It would be difficult to represent complexly structured events in this format. Parsing of CSV files is, however, very efficient due to its simple format. Each data record in a CSV file occupies one line. Values on the line are separated by commas. An optional initial header line can contain column names, allowing CSV processing software to access values by name. It is important to note that whitespace is taken as part of values.

In the case where all events have the same number of arguments, and each position has the same interpretation across lines, the CSV format is optimal. As an example consider the CSV file containing drawing commands taking x- and y-coordinates as arguments:

```
command, x, y
move, 3, 4
draw, 0, 4
move, 0, 0
draw, 3, 4
```

However, the typical case is that different commands take different numbers and kinds of arguments, with different interpretations. During the latest RV competition CSV files were for example generated for keeping track of Java operations on maps (update map,

[8] See `http://www.ietf.org/rfc/rfc4180.txt`

create the collection of keys of the map, create a derived iterator from this key set, use the iterator - and check that after an update to a map no such derived iterator is further used). The CSV files used a header as in the following example:

```
event, map, collection, iterator
updateMap, 6750210, ,
createColl,6750210, 2081191879,
createIter, , 2081191879, 910091170
useIter, , , 910091170
updateMap, 1183888521, ,
```

Note that in order to match the header, empty fields are required when a particular column is not relevant to an event. A CSV file can also be constructed without using a header, in which case the CSV file reading software must know and interpret the postions of the arguments correctly. Here arguments are just listed sequentially without blank fields in between. Positions here have different interpretations for different commands, for example argument number 1 in a updateMap command is a map while argument number 1 in a createIter command is a collection:

```
updateMap, 6750210
createColl, 6750210, 2081191879
createIter, 2081191879, 910091170
useIter, 910091170
updateMap, 6750210
```

Finally one can consider a format where there is also no header, but where fields are named in each row (every second position in a line is the name of a value, which then follows in the next position):

```
updateMap, map, 6750210
createColl, map, 6750210, collection, 2081191879
createIter, collection, 2081191879, iterator, 910091170
useIter, iterator, 910091170
updateMap, map, 6750210
```

eXtended Markup Language (XML). This standardised[9] format associated with web services is a markup language where data are tagged. In the RV competition five tags were introduced: `log`, `event`, `name`, `field`, and `value`. The following gives a log consisting only of the second event in the previous log.

```
<log>
  <event>
    <name>createColl</name>
    <field>
      <name>map</name>
      <value>6750210</value>
    </field>
```

[9] See `https://www.w3.org/TR/REC-xml`

```
      <field>
        <name>collection</name>
        <value>2081191879</value>
      </field>
    </event>
  </log>
```

This format clearly supports structured data and also metadata, for example an event could be further tagged with additional time information:

```
<event timestamp="1462810918">
```

Although this does not add much functionality as the same information could be represented as data (e.g. in CSV); the role of metadata is to separate data from information that describes it.

However, the above representation is verbose. Similar to the CSV format, the XML format can be simplified, not mentioning names of fields, but rather giving them just by position. The above one-event log would in such a solution become simpler, although still somewhat verbose:

```
<log>
  <event>
    <name>createColl</name>
    <value>6750210</value>
    <value>2081191879</value>
 </event>
</log>
```

Finally, XML supports the notion of *schema* that defines the expected structure and can be used to validate XML documents.

JavaScript Object Notation (JSON). This standardised[10] format stores structured attribute-value pairs as well as arrays. The first one-event log presented as XML above can be captured in JSON as follows.

```
[
  {
    "createColl" : {
      "map" : "6750210",
      "collection" : "2081191879"
     }
  }
]
```

This seems to have the advantages of XML but is relatively more concise. Similar to the cases of CSV and XML, JSON can be made more succinct by using arrays to model positional arguments, as in the following where we have listed all the events in the original CSV file:

[10] See `https://tools.ietf.org/html/rfc7159`

```
[
  {"updateMap" : ["6750210"]},
  {"createColl" : ["6750210", "2081191879"]},
  {"createIter" : ["2081191879", "910091170"]},
  {"useIter" : ["910091170"]},
  {"updateMap" : ["6750210"]}
]
```

Tool Formats. Finally, we note that some tools have their own propriety trace file format. Monpoly [14] has a plain text format with an event per-line where each line contains a timestamp, event name and then (optionally) some data parameters. Both OCLR-Check [28] and BeepBeep [35] make use of custom XML formats.

6 Discussion and Conclusion

The aim of this paper was to review various notions of trace from a runtime verification perspective, and beyond. This review has been relatively lightweight but hopefully provides some discussion points and useful references. We finish with a few discussion points relevant to the topic of traces, which we have not yet touched.

Rolling Logs. Something that is rarely dealt with in runtime verification is the issue of monitoring logs from systems that have been running for a long time. The first problem is that one will not want to repeatedly analyze all logs from the beginning of time. Approaches will be needed for *accumulating* monitoring results from past analyses. The second problem is that of *bootstrapping*, if one has not been recording logs so far, but wants to start monitoring, then it may be necessary to make some assumptions about the unseen logs.

Uncertainty. A common issue is where a trace contains partial information. This can either be partial by construction and therefore known to be partial, for example where a system's execution is sampled only periodically for efficiency reasons [13, 40, 67]. Or it may be partial due to unreliability in the recording process. In either case it may be necessary to either estimate or predict what has been omitted, or provide some confidence in the computed verdict. An alternative approach is to compute the *distance* between the observed behaviour and expected behaviour [55, 57].

Concurrency. We have ignored the issue of concurrent or distributed systems. Such systems can be abstracted as a set of separate but related execution traces with a single trace per concurrent or distributed process. A notable property of such systems is that there is (generally) no notion of a *global clock*. If the behaviour of each process is independent then it may be sufficient to consider the behaviour of the system as a set of independent sequential traces. However, it is more common for there to be dependencies between processes. In such a case there is a choice: one can enforce a *total ordering* of events or record the observed *partial order*.

The first case equates to flattening the set of traces into a single trace, for example serialising the trace by selecting an arbitrary ordering. But unless this ordering is enforced (via synchronization) the trace may not reflect actual behaviour, possibly leading to false positives or false negatives in monitoring. If the ordering is enforced by synchronization this can have a large impact on performance.

In the second case it is typical [50, 53, 63, 64] to consider a *distributed computation* as a *partial order* $\langle E, \rightarrow \rangle$ on a set of events E based on the *happens-before* relation $\rightarrow$. The happens-before relation necessarily totally orders events from the same process and represents synchronisation (e.g. message passing) between different processes. In such a setting a *global state* (or *consistent cut*) is a tuple of events (one from each process) representing a frontier in the distributed computation that satisfies the happens-before relation. These global states can then be formed into a *computation lattice* or *state lattice* where one global state is above another global state if it occurs strictly later in the computation. This differs from the previous structure as the nodes now represent global states rather than individual events. A path through the computation lattice is one possible global trace of the system and it is typical to consider all such traces. In the context of multithreaded programs there exist methods generalising the happens-before relationship by considering additional causal relationships [38, 65]. This can lead to more possible global traces being explored, increasing the chances of finding buggy behaviour even if this behaviour was not observed at runtime. Such exploration of all possible global traces in the partial order has similarities to model checking [37]. However, a model checker will explore all possible traces of the program, whereas the above described method will not.

As a concluding remark, it is clear that any efforts to unify notions of trace and standardise how runtime verification tools record and process events will be beneficial to both the developers and users of such tools.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
2. Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
3. Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994.
4. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.
5. AspectC++. Aspect oriented programming for C++. `http://www.aspectc.org`, 2016.
6. AspectJ. Aspect oriented programming for Java. `https://eclipse.org/aspectj/`, 2016.
7. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.

8. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
9. Howard Barringer and Klaus Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.
10. Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, chapter First International Competition on Software for Runtime Verification, pages 1–9. Springer International Publishing, Cham, 2014.
11. Ezio Bartocci, Borzoo Bonakdarpour, Yliès Falcone, Christian Colombo, Normann Decker, Felix Klaedtke, Klaus Havelund, Yogi Joshi, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2016 (To appear).
12. Ezio Bartocci, Luca Bortolussi, Dimitrios Milios, Laura Nenzi, and Guido Sanguinetti. *Hybrid Systems Biology: Fourth International Workshop, HSB 2015, Madrid, Spain, September 4-5, 2015. Revised Selected Papers*, chapter Studying Emergent Behaviours in Morphogenesis Using Signal Spatio-Temporal Logic, pages 156–172. Springer International Publishing, Cham, 2015.
13. Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, chapter Adaptive Runtime Verification, pages 168–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
14. David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monpoly: Monitoring usage-control policies. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2012.
15. David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
16. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, May 2015.
17. Johan Bengtsson and Wang Yi. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, chapter Timed Automata: Semantics, Algorithms and Tools, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
18. Ian Cassar and Adrian Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014.*, pages 54–68, 2014.
19. Ian Cassar and Adrian Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *International Conference on integrated Formal Methods (iFM)*, 2016.
20. Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *TACAS '09*, pages 246–261, Berlin, Heidelberg, 2009.
21. Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, chapter Parametric Runtime Verification of C Programs, pages 299–315. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

22. CIL. C Intermediate Language. https://www.cs.berkeley.edu/~necula/cil/, 2016.
23. Christian Colombo, Adrian Francalanza, and Rudolph Gatt. *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, chapter Elarva: A Monitoring Tool for Erlang, pages 370–374. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
24. Normann Decker, Martin Leucker, and Daniel Thoma. *NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, chapter jUnitRV–Adding Runtime Verification to jUnit, pages 459–464. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
25. Normann Decker, Martin Leucker, and Daniel Thoma. *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, chapter Monitoring Modulo Theories, pages 341–356. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
26. Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3):16:1–16:30, April 2009.
27. Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. *Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, chapter On Temporal Logic and Signal Processing, pages 92–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
28. Wei Dou, Domenico Bianculli, and Lionel Briand. Oclr: A more expressive, pattern-based temporal extension of ocl. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications - Volume 8569*, pages 51–66, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
29. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In Manfred Broy and Doron Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
30. Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second International Competition on Runtime Verification. In *6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, volume LNCS, page 16, Vienne, Austria, September 2015. Springer.
31. E. A. Gol, E. Bartocci, and C. Belta. A formal methods approach to pattern synthesis in reaction diffusion systems. In *53rd IEEE Conference on Decision and Control*, pages 108–113, Dec 2014.
32. Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, chapter Runtime Verification Based on Register Automata, pages 260–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
33. Radu Grosu, Scott A. Smolka, Flavio Corradini, Anita Wasilewska, Emilia Entcheva, and Ezio Bartocci. Learning and detecting emergent behavior in networks of cardiac myocytes. *Commun. ACM*, 52(3):97–105, March 2009.
34. Iman Haghighi, Austin Jones, Zhaodan Kong, Ezio Bartocci, Radu Gros, and Calin Belta. Spatel: A novel spatial-temporal logic and its applications to networked systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 189–198, New York, NY, USA, 2015. ACM.
35. S. Halle and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, April 2012.

36. Klaus Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
37. Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
38. Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. *SIGPLAN Not.*, 49(6):337–348, June 2014.
39. S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković. From signal temporal logic to FPGA monitors. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 218–227, Sept 2015.
40. Kenan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott D. Stoller, and Radu Grosu. *Runtime Verification: 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, chapter Runtime Verification with Particle Filtering, pages 149–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
41. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.
42. MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
43. Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *International Conference on Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 167–182. Springer, September 2013.
44. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
45. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
46. Hong Lu and Alessandro Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007-99, Microsoft Research, August 2007.
47. Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Serbanuta, and Grigore Rosu. RV-monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 14th International Conference on Runtime Verification (RV'14)*. LNCS, September 2014.
48. Oded Maler and Dejan Nickovic. *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systmes, FORMATS 2004, and Formal Techniques in Real-Time and Fault -Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004. Proceedings*, chapter Monitoring Temporal Properties of Continuous Signals, pages 152–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
49. Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
50. M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 494–503, May 2015.
51. S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. RiTHM: a tool for enabling time-triggered runtime verification for c programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 603–606, 2013.
52. Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.
53. Vinit A. Ogale and Vijay K. Garg. *Detecting Temporal Logic Predicates on Distributed Computations*, pages 420–434. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

54. Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 1–13, Berlin, Heidelberg, 2008. Springer-Verlag.
55. Fabrizio Pastore and Leonardo Mariani. AVA: supporting debugging with failure interpretations. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 416–421, 2013.
56. R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481–491, Nov 2008.
57. Giles Reger. Suggesting edits to explain failing traces. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 287–293, 2015.
58. Giles Reger and David E. Rydeheard. From first-order temporal logic to parametric trace slicing. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 216–232, 2015.
59. Adam Renberg. Test-inspired runtime verification. Master's thesis, Royal Institute of Technology (KTH), Stockholm, 2014.
60. A. Russ. Detecting security incidents using windows workstation event logs. Technical report, Sans Institute InfoSec Reading Room, 2013.
61. Luc Segoufin. *Computer Science Logic: 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006. Proceedings*, chapter Automata and Logics for Words and Trees over an Infinite Alphabet, pages 41–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
62. K. Selyunin, T. Nguyen, E. Bartocci, D. Nickovic, and R. Grosu. Monitoring of MTL specifications with IBM's spiking-neuron model. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 924–929, March 2016.
63. Alper Sen and Vijay K. Garg. Rv '2003, run-time verification (satellite workshop of cav '03) partial order trace analyzer (pota) for distributed programs. *Electronic Notes in Theoretical Computer Science*, 89(2):22 – 43, 2003.
64. Alper Sen and Vijay K. Garg. *Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing*, pages 171–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
65. Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. *Maximal Causal Models for Sequentially Consistent Systems*, pages 136–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
66. Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Interaspect: aspect-oriented instrumentation with GCC. *Formal Methods in System Design*, 41(3):295–320, 2012.
67. Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, chapter Runtime Verification with State Estimation, pages 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
68. Volker Stolz. Temporal assertions with parametrized propositions*. *J. Log. and Comput.*, 20:743–757, June 2010.
69. Tim Todman, Stephan Stilkerich, and Wayne Luk. In-circuit temporal monitors for runtime verification of reconfigurable designs. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 50:1–50:6, New York, NY, USA, 2015. ACM.
70. C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, October 2007.