Runtime Verification
Past Experiences and Future Projections

Klaus Havelund*[1], Giles Reger[2], and Grigore Roşu[3]

[1] Jet Propulsion Laboratory, California Institute of Technology
[2] University of Manchester, Manchester, UK
[3] University of Illinois at Urbana-Champaign

Abstract. The paper provides an overview of the work performed by the authors since the year 2000 in the field of *runtime verification*. Runtime verification is the discipline of analyzing program/system executions using rigorous methods. The discipline covers such topics as specification-based monitoring, where single executions are checked against formal specifications; predictive runtime analysis, where properties about a system are predicted/inferred from single (good) executions; fault protection, where monitors actively protect a running system against errors; specification mining from execution traces; visualization of execution traces; and to be fully general: computation of any interesting information from execution traces. The paper attempts to draw lessons learned from this work, and to project expectations for the future of the field.

1 Introduction

Runtime verification (RV) [10, 32, 41] has emerged as a field of computer science within the last couple of decades. RV is concerned with the rigorous monitoring and analysis of software and hardware system executions. The field, or parts of it, can be encountered under several other names, including, e.g., runtime checking, monitoring, dynamic analysis, and runtime analysis. Since only single executions are analyzed, RV scales well compared to more comprehensive formal methods, but of course at the cost of coverage. Nonetheless, RV can be useful due to the rigorous methods involved.

The first and last author's initial interest in RV started around 2000. We had at that time explored software model checking with the Java PathFinder tool [43, 49]. Part of that work focused on exploring the spectrum from full formal verification to more scalable testing. That investigation led to our interest in RV. Our initial efforts were inspired by Doron Drusinky's Temporal Rover system [30] for monitoring temporal logic properties, and by the company Compaq's work on predictive data race and deadlock detection algorithms [36]. These algorithms can detect the *potential* for a data race or deadlock by analyzing a run that does

* 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.

not necessarily encounter the error. This paper reports on our own RV work, with some references to related work that specifically inspired us or which we find closely related, and discusses the lessons learned and our perspective on the future of this field.

A particular software or hardware system to be monitored is from here on referred to as the System Of Interest (SOI). We shall, due to our own lack of experience in monitoring hardware systems, limit our focus to monitoring of software systems, although for the majority of the discussion this distinction is not important. An important part of RV is how to extract an execution trace from an SOI, for example through manual logging or automated code instrumentation. This touches on the combination of static and dynamic analysis. We are not dealing with how to obtain various executions, as in e.g. test case generation (another important topic covered e.g. in [18] in this volume). Runtime verification can be used prior to deployment for testing purposes, referred to as test oracles in [18], and during deployment for ensuring safety and security, e.g. as part of a fault protection strategy.

As a more formal account, assume an SOI S, and assume that an execution of S is captured as an execution trace $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$, which is a sequence of observed events. Each event e_i captures a snapshot of S's execution state. Monitors can be deeply embedded in the running system, able to access the full state of the system, or they can observe from a "distance", receiving execution events (data records) from the running system. Assume the type $\mathbb{E}$ of events, then the RV problem can be formulated as constructing a program $M : \mathbb{E}^* \to D$, which when applied to a trace σ, as in $M(\sigma)$, returns some data value $d \in D$ in a domain D of interest. The problem can be generalized to computing a result from multiple traces (as e.g. done in learning and statistical model checking), giving M the type[4] $M : \mathcal{P}(\mathbb{E}^*) \to D$.

In specification-based RV, M can be generated from a formal specification given in e.g. temporal logic, state machine notation, regular expressions, and d is a value in the Boolean domain ($d \in \mathbb{B}$), or some extension of the Boolean domain as discussed in [12], indicating whether the trace conforms to the specification. However, the field should be perceived broadly, e.g. d can be a visualization of the execution trace, a learned specification (specification mining), statistical information about the trace, an action to perform on the running system S, etc.

The body of the paper is largely organized according to the time periods in which the research was performed. Section 2 describes the first systems we developed, starting with monitoring propositional events, and transitioning to monitoring of parametric events carrying data, focusing on expressive logics as well as efficient monitoring algorithms based on trace slicing. Section 3 describes our experiments with aspect-oriented programming as a natural way of combining RV and code instrumentation. Section 4 describes early rule-based systems, as well as systems developed specifically targeting space mission applications. Section 5 describes our experiments with internal DSLs defined as APIs in a programming language. Furthermore, trace slicing is yet again pursued for an

[4] For any set S, $\mathcal{P}(S)$ is the power set of S, containing all subsets of S as elements.

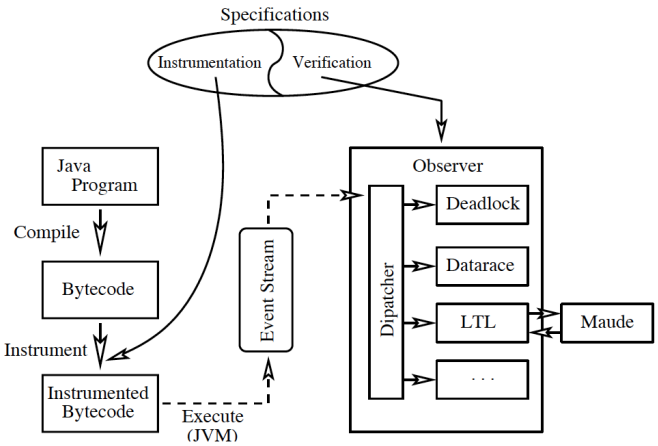

Fig. 1. The JPaX architecture.

expressive logic, and a system for Complex Event Processing (CEP) is developed, where the result of monitoring is a more complex data structure than just a Boolean value. Section 6 covers mostly the entire period, and describes efforts in predictive analysis, concerned with predicting anomalies in programs from successful observed executions. Finally, Section 7 reflects on the presented work, and provides thoughts on the future of the field of runtime verification.

2 2000-2005 - From Propositional to Parametric RV

2.1 Java PathExplorer

Architecture Our first monitoring system, Java PathExplorer (JPaX) [48, 47] was a general framework for analyzing execution traces. It supported three kinds of algorithms: propositional temporal logic conformance checking, data race detection, and deadlock detection. Figure 1 shows JPaX's architecture. A Java program is instrumented (at byte code level) to issue events to the monitoring side, which is customizable, allowing the addition of new monitors. The temporal logic monitoring module was originally based on a propositional future time linear temporal logic, but was later extended to also cover past time.

Future time LTL The future time LTL monitoring used Maude to rewrite formulas. Consider, e.g., the LTL formula $p \,\mathcal{U}\, q$, meaning q eventually becomes true and until then p is true. The implementation of JPaX was based on classical equational laws for temporal operators, such as:

$$p \,\mathcal{U}\, q = q \wedge \bigcirc(p \,\mathcal{U}\, q) \qquad \text{and} \qquad \Box p = p \wedge \bigcirc(\Box p) \tag{1}$$

Consider the sample formula $\Box(green \rightarrow \bigcirc(\neg red \,\mathcal{U}\, yellow))$. Upon encountering a *green* in a trace, the formula will be rewritten into the following formula, which must be true in the next state: $(\neg red \,\mathcal{U}\, yellow) \wedge \Box(green \rightarrow$

($\neg red\ \mathcal{U}\ yellow$)). In Maude this was realized by a few simple rewrite rules, including the following two for the until operator (E is an event and T is a trace, the first rule handles the case of a trace consisting of only one event):

```
eq E |= X U Y = E |= Y .
eq E,T |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
```

Past time LTL Later, an efficient dynamic programming algorithm for monitoring *past time* logic was developed [47]. Consider the following past time formula: $red \rightarrow \blacklozenge green$ (whenever *red* is observed, in the past there has been a *green*). The algorithm for checking past time formulas like this uses two arrays, now and pre, recording the status of each sub-formula now and previously. Index 0 refers to the formula itself with positions ordered by the sub-formula relation. Then for this property, for each observed event the arrays are updated as follows.

```
bool pre [0..3], now [0..3];

fun processEvent(e) {              // Sub-formula:
  now[3] := (event = red)          // red
  now[2] := (event = green)        // green
  now[1] := now[2] || pre[1]       // PREV green
  now[0] := !now[3] || now[1]      // red -> PREV green
  if !now[0] then output(" property violated ");
  pre := now;
}
```

Data races and deadlocks When used for bug finding, the effectiveness of runtime verification depends on the choice of test suite. For concurrent systems this is critical, due to the many possible non-deterministic execution paths. *Predictive runtime analysis* approaches this problem by replacing a target property P with a stronger property Q such that there is a high probability that the program satisfies P iff a random trace of the program will satisfy Q. Some of the first such algorithms, which greatly inspired us, were implemented in Compaq's Visual Threads tool [36] for analyzing multi-threaded applications in C and C++. One such algorithm was the Eraser algorithm [68], for detecting *potentials* for data races (where two threads can access a shared variable simultaneously). It is often referred to as the *lock set* algorithm as each variable is associated with a set of locks protecting it. Alternatively, the *lock graph* algorithm, would detect "dining philosopher"-like deadlock potentials by building a simple lock graph where a cycle indicates a deadlock potential. We continued this line of work in a variety of ways. In [37] we explored the idea of letting a predictive analysis guide a model checker towards data race and deadlock potentials. In [15] we augment the original lock graph algorithm to reduce false positive in the presence of guard locks (locks that prevent cyclic deadlocks). Other forms of data races than those

detected by Eraser are possible. In [3] is described a dynamic algorithm for detecting so-called high-level data races (races involving collections of variables). Section 6 goes into more detail with research on predictive analysis.

2.2 Eagle

JPaX had a number of limitations. The perhaps most important was the propositional nature of the temporal logics. One could not, for example, monitor parametric events carrying data, such as *openFile("data.txt")*. A second drawback of JPaX was the separation between past time and future time temporal logic, in two different logical systems. More generally, it seemed to us unfortunate that one had to pick a particular logic amongst the many existing for writing temporal properties, including past and future time temporal logic, extended regular expressions, state machines, interval logics, real-time logics, data constraint logics, and statistical logics. It would be very attractive if a user could define his/her own temporal logic from a small set of primitives. These thoughts lead, during 2003, to the work on Eagle, first documented in [6]. Eagle was a small and general logic having similarities with the μ-calculus.

The logic allowed the definition of new temporal operators which could be parameterized with formulas and primitive data such as integers. In addition to the standard Boolean operators, the logic includes: $\bigcirc\ f$ (next f), $\odot\ f$ (previous f), $f_1\ \cdot\ f_2$ (concatenation: f_1 on part of the trace and f_2 on the remaining part of the trace), f (now f), and $\mathsf{N}(f_1\ ,...,f_n\ )$ (call N with arguments). A fundamental idea in Eagle was the option for a user to define temporal operators using recursion similar to the equations in (1) on page 3. Such user-defined temporal operators are defined as follows in Eagle:

min Until(**Form** f_1 , **Form** f_2) = $f_2 \vee (f_1\ \wedge \bigcirc$ Until(f_1 ,f_2))
max Always(**Form** f) = f $\wedge \bigcirc$ Always(f)

Note how the different operators are defined as respectively minimal and maximal fixpoints, reflecting the definition of liveness and safety properties respectively. The difference in semantics appears at the boundaries of a trace where remaining minimal terms evaluate to false whereas maximal terms evaluate to true. These can now be used in writing monitors as follows:

mon M = Always(x > 0 $\Rightarrow$ Eventually(y > 0))

Eagle handles data parameterized formulas through data parameterized rules. Consider the first-order temporal logic formula ("whenever $x > 0$, then if we name x's value k, then eventually $y = k$"): $\Box(x > 0 \rightarrow \exists k\ .\ (x = k\ \wedge\ \Diamond y = k))$. This can be formulated in Eagle using a data parameterized rule as follows.

min yBecomes(**int** k) = Eventually(y = k)
mon M = Always(x > 0 $\Rightarrow$ yBecomes(x))

The later Hawk system [27] was an attempt to tie Eagle to the monitoring of Java programs with automated code instrumentation using aspect-oriented programming, specifically AspectJ [57]. A similar (and simultaneous) integration of

parametric runtime verification (with LTL) and AspectJ was presented in the J-LO tool [78]. Hawk supports two modal constructs inspired by dynamic logic: the construct <e> F means that e *can* occur and the proposition F is true thereafter. The construct [e] F means that *if* e occurs, then F is true thereafter. As a complete example, consider the following observer, monitoring that elements put into a buffer eventually get taken out of the buffer:

```
observer BufferObserver {
  var Buffer b ; var Object o ; var Object k ;

  mon B = Always ([b?.put(o?)]
              Eventually ( <b.get() returns k?> (o == k) )) .
}
```

2.3 JavaMOP

The same JPaX limitations that motivated the development of Eagle also stimulated the apparition of monitoring-oriented programming (MOP) [22, 21, 23]. MOP proposed that runtime monitoring be supported and encouraged as a fundamental principle of software development, where monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program. Violations and/or validations of specifications can trigger user-defined code at any points in the program, in particular recovery code, outputting/sending messages, or raising exceptions. MOP has made three important early contributions. First, it proposed specification formalism independence, allowing users to insert their favorite or domain-specific requirements specification formalisms via *logic plugin* modules. Second, it proposed automated code instrumentation as a means to weave the monitoring checking code within the application; the first version in 2003 used Perl for instrumentation [22], while the subsequent versions starting with 2004 [21] used AspectJ [57]. Finally, it proposed a formalism-independent semantics and implementation for parametric specifications.

Parametric properties are properties with free variables, allowing us to describe behaviors of collections of related objects. Consider, for example, the following JavaMOP parametric property.

```
SafeLock(Lock l, Thread t) {
  event acquire before(Lock l, Thread t):
    call( * Lock.acquire()) && target(l) && thread(t) {}
  event release before(Lock l, Thread t):
    call( * Lock.release()) && target(l) && thread(t) {}
  event begin before(Thread t):
    execution( * *.*(..)) && thread(t) && !within(Lock+) {}
  event end after(Thread t):
    execution( * *.*(..)) && thread(t) && !within(Lock+) {}

  cfg: S -> S begin S end | S acquire S release | epsilon
```

```
    @fail { System.out.println("Improper lock usage"); }
}
```

It has two parameters: a lock and a thread. The four event declarations declare the parametric events of interest, and the property, in this case formalized using the context-free grammar (CFG) plugin, states that each acquire and release event should be paired in the same method. Any mismatched acquire or release is considered to be a violation of the property. At violation we chose to report an error message, but any Java code can be executed, e.g., recovery code. Note that this property cannot be expressed using regular patterns or automata.

It is not trivial to monitor parametric properties efficiently. For the example it is not uncommon in a multi-threaded Java program execution to see thousands of threads created/terminated and thousands of synchronization locks acquired/released by such threads dynamically. Conceptually, execution traces are sliced according to each observed instance of the parameters, and each slice is checked by its own monitor instance in a manner that is independent of the employed specification formalism. The practical challenge is how to deal with the potentially huge number of monitor instances.

JavaMOP proposed several optimizations, presented in [66] together with the mathematical foundations of parametric monitoring. For example, we can ignore parameter instances that can never reach the target monitor states (e.g., not all threads use all locks). Also, some monitors can become unnecessary during execution because the objects that can generate the triggering events have died; such unnecessary monitors can and should be garbage collected.

A demo of JavaMOP is found at http://fsl.cs.uiuc.edu/JavaMOPDemo.html. The academic JavaMOP project has been migrated into the commercial RV-Monitor tool at http://runtimeverification.com/monitor. In addition to efficient support for simultaneous monitoring of multiple specifications, a major innovations of RV-Monitor is to separate instrumentation from the efficient monitoring data-structures. The former can be done either manually or using AspectJ (statically at compile time or dynamically as a Java agent), while the latter is automatically generated as a library from the parametric specifications.

3 2005-2006 - Further Experimentation with AOP

Whilst initial runtime verification frameworks targeted Java, the RMOR (Requirement Monitoring and Recovery) framework [38] targeted the monitoring of C programs against state machines using a homegrown aspect-oriented framework to perform program instrumentation. RMOR is implemented in OCaml using CIL (C Intermediate Language), a C program analysis and transformation system, itself written in OCaml. Consider as an example an application for uplinking data from a planetary rover to a space craft, and consider the property: *"It is illegal to have more than one connection opened at any time"*. This requirement can be formulated as follows.

```
monitor UplinkRequirement {
  event OPEN = after call(main.c:open_connection);
  event CLOSE = after call(main.c:close_connection);

   initial state Closed {
     when OPEN → Opened;
  }

   live state Opened {
    when CLOSE → Closed;
    when OPEN → error;
  }
}
```

The `Opened` state is a *live* state as indicated by the modifier keyword `live`, meaning a non-acceptance state. Other state modifiers include super states as in hierarchical state charts. It is possible to provide a call-back handler function to be called for each detected violation. However, RMOR is propositional.

In previous solutions (such as Hawk and MOP) we have seen monitors translated *to* aspects. A more radical approach is to take the view that monitors *are* aspects. Some of our experiments went in the direction of what today is called *state-full aspects* [80, 1]. We proposed this line of work already in [34]. An (non-finished) attempt in this direction was XspeC [50], designed to be an extension of ACC (an aspect-oriented programming framework for C) with data parameterized monitoring using state machines. As an example, consider the property of a C program that a file should be opened and eventually closed in that order. When an already opened file is re-opened the attempt should be logged and when the program terminates all opened files should be closed. The specification in XspeC becomes as follows.

```
xspec OpenClose(char *file) {
  pointcut open : call(void openfile(char*)) && args(file);
  pointcut close : call(void closefile (char*)) && args(file);

  state FileClosed {
    after : open( file ) → FileOpen;
    after : close( file ) ⇒ error;
  }

  live state FileOpen {
    after : open( file ) ⇒ error { log( file ); }
    after : close( file ) → FileClosed;
    before : end { closefile ( file ); }
  }
}
```

The specification is parameterized with a file, meaning that it is intended to track the behavior of a file. The intended semantics is similar to the semantics of Tracematches [1] and MOP in that we consider a specification to denote an infinite set of monitors, one for each file as indicated by the parameter to the specification. The double arrow ($\Rightarrow$) denotes a transition that stays in the source state (for continued verification), in contrast to the single arrow ($\rightarrow$).

In [34] we discussed the idea (and similar work was proposed in HandlErr [74]), to extend aspect-oriented programming in two ways: vertically and horizontally. The pointcut languages originally supported, for example in AspectJ, have been limited, reducing to method calls and assignment to variables. A *vertical* extension consists of enriching the pointcut language to cover more concepts, such as e.g. branching on a conditional, cycling through a loop, or acquiring and releasing a lock. Some of the algorithms described in this paper analyzing multi-threaded programs for data races and deadlocks, for example, cannot use AspectJ for instrumentation since AspectJ does not support definition of pointcuts catching lock acquisitions and releases in the general case. In [17] we proposed extending AspectJ with new pointcuts: **lock**() and **unlock**(). A *horizontal* extension consists of changing the definition of advice to incorporate tracecuts. The ultimate extension of aspect-oriented programming is the product of a horizontal and a vertical extension. In addition, static analysis (theorem proving) can be invoked to prove stated properties. HandlErr e.g. allowed pre and post conditions, invariants in aspects.

A much later work presented in [73] is the InterAspect system, an aspect-oriented API in C for instrumenting C programs compiled with the GCC compiler infrastructure. InterAspect is implemented using the GCC plug-in API. The system allows for specification of tracecuts using regular expressions, much along the lines of MOP. InterAspect has access to GCC internals, which allows one to exploit static analysis during the weaving process. Consider the following file access property. Any access to a file object after the file has been closed is a memory error which might not manifest itself as incorrect behavior during testing. This can be formalized in InterAspect as the following "aspect" matching an execution as soon as any read is performed on a closed file.

```
tc = tc_create_tracecut (); tc_add_param(tc,"file", aop_t_all_pointer ());
tc_declare_call_symbol (tc,"open","(file)fopen()",AOP_AFTER);
tc_declare_call_symbol (tc,"read","fread(?,?,?,file)",AOP_BEFORE);
tc_declare_call_symbol (tc,"read_char","fgetc(file)",AOP_BEFORE);
tc_declare_call_symbol (tc,"close","fclose(file)",AOP_BEFORE);
tc_add_rule(tc,"open (read | read_char)* close (read | read_char)");
```

4 2006-2010 - Missions and Rules

4.1 Commanding and Monitoring

One project, described in [14], was driven by a collaboration between JPL and KSC (Kennedy Space Center) from where NASA's rocket launches take place.

The project had as a goal to develop a DSL for commanding and monitoring all aspects of a rocket launch platform in the moments up to a launch. The DSL was implemented as a Python API. A program would, through a publish-subscribe framework, command and monitor *items* distributed geographically across the KSC launch site. The state can be understood as a collection of *measurements*, representing data samples collected from sensors in the items, and distributed throughout the system on a message bus. Each measurement maps a variable name to a value. The DSL then provides a collection of functions for monitoring the state (collection of measurements) of the entire system as it evolves over time. From a temporal logic point of view, a trace is a sequence of collections of measurements. Some of these functions are shown below.

```
def verify (C, [R], [S]): ...
def verify_within (C, D, [R], [S]): ...
def verify_subset_within (N, C_list , D_list , [R], [S]): ...
def assert_constraint (S, C, R, [D], [F]): ...
def conditional_interrupt (S, C, R, [D], [F]): ...
```

The following symbols are used for arguments: C stands for a condition to be verified and R stands for a reaction to be executed in case a condition gets violated. Both C and R are assumed to be parameter-less functions. D stands for a duration, expressed in seconds. S stands for a string, generally a name associated with the verification operation for documentation purposes. N stands for a natural number. Finally, F stands for a Boolean flag indicating whether verification should be repeated in case of property violations. Arguments in square brackets [...] denote optional arguments (this is not Python syntax).

The functions (the first three of which are blocking, waiting for the verification to terminate) have the following meaning. verify verifies that the condition is true now. verify_within verifies that the condition C eventually becomes true within the time duration D. verify_subset_within verifies that at least N of a list of conditions become true within given durations, provided as a separate list matching in length. assert_constraint verifies that the condition is continuously true throughout the duration. conditional_interrupt is a variant of assert_constraint where if the condition at some point evaluates to *true*, the calling application is interrupted (temporarily stopped) while the reaction is executed. The DSL also provides functions for commanding items and interacting with users at terminals. The team at KSC subsequently developed a tabular DSL using spreadsheets, which is a form of external DSL built on top of the (internal) Python DSL.

4.2 RuleR.

RuleR [9] started life as a low-level event-based rule system into which other temporal specification languages were supposed to be compiled for efficient trace checking. The work was directly inspired by the complexity of the Eagle implementation. However, it then assumed a life of its own as a specification language. RuleR preserves the interest in monitoring data via parametric events but also

achieves high expressiveness through the use of powerful low-level features. The flavor of specifications in RuleR is different from those based on temporal logic seen earlier as they tend to be more *operational*. For example, to monitor the previous property $\Box(x > 0 \rightarrow \exists k \,.\, (x = k \ \wedge \ \Diamond y = k))$ we would monitor events `x` and `y` and whenever observing a relevant `x` event create an obligation to see a future `y` event with that value. This is captured by the following rule system.

```
ruler M {
  observes x(int), y(int);
  always start{ x(n:int) & n>0 -> wait(n); }
  state wait(n:int){ y(n) -> Ok; }
  forbidden wait;
  initials start;
}
```

This monitor declares a set of events being observed and then two rules. Rules are of the form

$$\mathit{conditions} \rightarrow \mathit{obligations}$$

and define *rewrite rules* on sets of *rule instances*. If the set of rule instances satisfy the conditions then the obligations should be applied to this set where an obligation may add or remove a rule instance from the set. Importantly, the only rules that can be applied are those that have a corresponding rule activation in the current set. This extends to data parameterization. If `wait(1)` is not in the current set then the event `y(1)` would not satisfy any conditions. Another aspect of a rule is its *modifier*. In the above example the **always** modifier means that a rule activation should be kept if its corresponding rule is applied to it, whilst the **state** modifier indicates that it should be removed. The following evaluation illustrates the above rule system applied to a sequence of events.

$$\{start\} \xrightarrow{x(5)} \underbrace{\{start, wait(5)\}}_{A} \xrightarrow{y(5)} \underbrace{\{start\}}_{B} \xrightarrow{x(1)} \{start, wait(1)\} \xrightarrow{end} \bot$$

The final result is failure ($\bot$) as the `wait` rule is in the **forbidden** set, which means that a trace ending with one of these rules in its set of rule activations is not accepted. RuleR was given a finite-trace semantics with four verdicts. The verdicts STILL_TRUE and STILL_FALSE are given if the rule system would accept/reject the trace if it were to end at the current event, whilst the verdicts TRUE and FALSE were reserved for traces where every extension would be accepted/rejected. For the above example, the A set of rule instances would be given the verdict STILL_FALSE whilst the B set would be given STILL_TRUE. These multiple verdicts support various translations of finite-trace linear temporal logics.

A more realistic example is the following rule system checking the proper usage of Java iterators. Here the **assert** keyword requires that at least one of the given rules is applied on each step. This allows, for example, the rule system to detect failure on the event sequence consisting only of a `next` event.

```
ruler SafeIteratorCheck{
```

```
  observes hasNext(obj), next(obj);
  always Start{ hasNext(i:obj) -> Next(i); }
  state Next(i:obj){ next(i) -> Ok; }
  assert Start, Next;
  initials Start;
}
```

RuleR allowed for very complex rule systems that could be *chained* together such that one rule system produced outputs for another rule system to consume as input events. Rule systems could be combined sequentially, in parallel, and conditionally. Another powerful feature was the use of non-determinism and rules as data. However, it was difficult to find a practical need for such features.

4.3 LogScope

A project solidly rooted in an actual space mission was the development of the LogScope temporal logic for log analysis [7]. The purpose of the project was to assist the team testing the flight software for JPL's Mars rover Curiosity, which successfully landed on Mars on August 6, 2012. The software produces rich log information. Traditionally, these logs are analyzed with complex Python scripts. The LogScope logic was developed to support notions more comprehensible to test engineers, including a very simple and convenient data parameterized temporal logic, which was translated to a form of data parameterized automata, which themselves can be used for specification of more complex properties that the temporal logic cannot express. LogScope was furthermore implemented in Python, allowing Python code fragments to be included in specifications, all in order to integrate with the existing Python scripting culture at JPL.

As an example, consider the property *"Whenever a flight software power command is issued, before the next flight software command there should follow a dispatch of that command on board, and then exactly one success of that command within 5 seconds. Before the dispatch there should be no dispatch failure, and in between the dispatch and the success there should be no execution failure"*. Commands have names x and numbers y. This property can be specified as follows in LogScope:

```
pattern Commands :
  COMMAND{Type:"FSW", Name:x, Num:y} where {: x.startswith("PWR") :} ⇒
    [
      !EVR{DispatchFailure:x, Num:y},
       EVR{Dispatch:x, Num:y, Time: t1},
      !EVR{Failure:x, Num:y},
       EVR{Success:x, Num:y, Time: t2} where {: t2 − t1 <= 5 :},
      !EVR{Success:x, Num:y}
    ] upto COMMAND{Type: "FSW"}
```

A specification consists of one or more specification units, each of which is either a temporal logic *pattern* (as above), or a parameterized *automaton*. A pattern

has a name, and is triggered by an event. When the event is observed in the log, the consequence must be observed, optionally up to some other event, which then limits the scope of the pattern. The consequence can be that an event must eventually occur, or not occur, or it can be a list of consequences, enclosed in either square brackets (as here) indicating the consequences must occur in that order, or curly brackets (not shown) indicating that the consequences must occur but any order is allowed. Note the lack of temporal operators as found in classical LTL. The **where**-clauses can contain Python expressions inside {: . . . :} brackets. The formula reflects the linear ordering of a time line [75], but textually presented. In general the user can define Python functions at the beginning of a specification file to be used in such predicates.

LogScope also allows testers to write properties as parameterized automata, to which the temporal patterns are also translated. Just as events can be parameterized with values, so can states. Automata can furthermore be visualized, which has shown to be useful for creators of patterns to confirm their meaning. The automaton for pattern Commands above is the following.

```
automaton Commands {
  always S1 {
    COMMAND{Type:"FSW", Name:x, Num:y}
      where {: x.startswith("PWR") :} ⇒ S2(x,y)
  }
  hot state S2(x,y) {
    EVR{DispatchFailure:x, Num:y} ⇒ error
    EVR{Dispatch:x, Num:y, Time: t1} ⇒ S3(x,y,t1)
  }
  hot state S3(x,y,t1) {
    EVR{Failure:x, Num:y} ⇒ error
    EVR{Success:x, Num:y, Time:t2} where {: t2 − t1 <= 5 :} ⇒ S4(x,y)
  }
  state S4(x,y) {
    EVR{Success:x, Num:y} ⇒ error
  }
}
```

5 2010-2017 - Internal DSLs, Slicing, and CEP

5.1 TraceContract

TraceContract [8] is an internal Scala DSL (effectively an API) for monitoring, based on a mixture of temporal logic and state machines. TraceContract, although a research tool, was used for analysis of command sequences sent to NASA's LADEE (Lunar Atmosphere and Dust Environment Explorer) spacecraft throughout its mission. Consider the LogScope specification on page 12. In order to specify this property in TraceContract we first define the event kinds, for example as follows:

```
trait Event
case class Command(time:Int,kind:String,name:String,nr:Int) extends Event
case class DispatchFailure(time:Int,name:String,nr:Int) extends Event
case class Dispatch(time:Int,name:String,nr:Int) extends Event
case class Failure(time:Int,name:String,nr:Int) extends Event
case class Success(time:Int,name:String,nr:Int) extends Event
```

Events are commonly modeled as objects (instances) of **case** classes (A **case** class allows pattern matching against its objects), all extending the Event trait (similar to abstract class in Java). Each event type is parameterized with data (the constructor parameters), which must be provided when creating an object of the class. The following monitor corresponds to the LogScope monitor on page 12, but now expressed in the internal Scala DSL.

```
class Commands extends Monitor[Event] {
  require {
    case Command(_, "FSW", x, y) if x.startsWith("PWR") ⇒
      hot {
        case DispatchFailure(_, `x`, `y`) ⇒ error
        case Dispatch(t1, `x`, `y`) ⇒ hot {
          case Failure(_, `x`, `y`) ⇒ error
          case Success(t2, `x`, `y`) if t2 - t1 <= 5 ⇒
            state { case Success(_, `x`, `y`) ⇒ error }
        }
      } upto { case Command(_,"FSW",_,_) ⇒ true }
  }
}
```

Our property is defined as a class Commands extending the class Monitor, which is parameterized with the event type, and which defines all the TraceContract DSL functions (marked in blue) and constants (marked in red). The DSL functions in this example all take as argument a Scala partial function enclosed in curly brackets, and defined with **case** statements.

The call of the function require (when a Commands object is created) causes a side-effect, namely storing the property represented by the partial function. Note that quotes around names, as in `x` means: match the value previously bound to x. The underscore '_' is the wildcard pattern that always matches. The monitor can be instantiated and applied to a trace (a list of events). TraceContract offers numerous additional constructs, including other kinds of anonymous states (e.g. strong next), state machines with named states, linear temporal logic, and the possibility to combine these with Boolean combinators (and, or, not). Mixed with general Scala programming this becomes a very powerful paradigm. A simpler version of TraceContract, but making states queryable facts (useful for expressing past time properties), is presented in [39].

A few other internal runtime verification DSLs/APIs have been developed. For example, a propositional Haskell DSL for linear temporal logic [79], and a Java API re-implementing MOP's trace slicing algorithms [16].

5.2 LogFire

Another example of an internal Scala DSL is LogFire [40]. LogFire is a rule-based system similar to RuleR, but based on the Rete algorithm implemented in several rule-based systems. LogFire was part of an investigation of the Rete algorithm's applicability for runtime verification. The algorithm maintains a network of facts to avoid re-evaluating all conditions in each rule's left-hand side each time the fact memory changes. We modified the Rete algorithm in a couple of ways to fit the runtime verification objective, including an indexing optimization and introducing the distinction between events and facts. As an example of a rule-system in LogFire consider safe use of Java iterators, where hasNext must be called before any call of next. This property can be formalized in LogFire as follows.

```
class HasNext extends Monitor {
  val hasNext, next = event
  val Safe = fact

  "r1" – hasNext('i) ↦ insert (Safe('i))
  "r2" – Safe('i) & next('i) ↦ remove (Safe)
  "r3" – next('i) & not(Safe('i)) ↦ error
}
```

As in TraceContract, a monitor is defined as an extension of a class Monitor, which defines the LogFire DSL features. The first two lines define the events that are observed and the facts (Safe) that the rules will generate. The monitor contains three named rules. Each rule has the form:

"name" – $cond_1(...)$ & ... & $cond_n(...)$ ↦ $action$

starting with a name (a string value), a conjunction of conditions, and an action to execute (following the ↦ symbol) in case the conditions evaluate to true. The insert function adds a new fact to the fact database, and the function remove (id) removes the fact id referred to on the left-hand side of the rule. The specification should be self-explanatory. In [40] it is described how higher-level operators can be defined in a few lines of code, generating rules automatically.

5.3 QEA

Quantified event automata (QEA) [5] and the associated MarQ tool [65] were introduced to take advantage of the efficient trace slicing approach previously introduced in the JavaMOP tool [63] (see Section 2.3) whilst dealing with some of the limitations with respect to expressiveness. QEA consist of a list of quantifications and an automaton. Consider the following example specification of the command property given on page 12. The specification begins with universal quantification over the command name and number and then gives an automaton structure similar to that of the LogScope monitor but the underlying semantics are quite different.

```
qea(Commands){
  forall (name, number)
  accept skip(1){
    command(name,number) → 2
  }
  next(2){
    dispatchFailure (name,number) → Fail
    dispatch (name,number,t1) → 3
  }
  next(3){
    failure (name,number) → Fail
    success (name,number,t2) if t2−t1 ≤ 5 → 4
  }
  accept skip(4){
    success (name,number,t) → Fail
  }
}
```

The semantics is defined in terms of *slicing* with respect to the quantified variables. For a given name and number pair an input trace is projected to preserve only events relevant to those values, giving a so-called *trace slice*. This trace slice is checked with respect to the given automaton. This semantics allows for efficient indexing structures that lookup the relevant part of the monitoring data to update given an event. However, to make the above slicing framework work incrementally is non-trivial as the values with which the trace is to be sliced are being discovered as the slice is being observed. The QEA work formalizes the notion of acceptance using quantification and extends[5] the framework to allow for *existential quantification* and *local state* via *unquantified/free variables*. The two specifications below demonstrate these features.

```
qea(RoverCommand){
  forall (q) exists(s) forall (r)
  accept skip( start ) {
    declare (q, r) → inside
  }
  skip( inside ) {
    ping(r, s) → pinged
  }
  skip(pinged){
    ack(s, r) → Success
  }
}
```

```
qea(AuctionBidding) {
  forall ( i)
  accept next(start){
    list (i , r) do c := 0 → listed
  }
  accept next( listed ){
    bid(i , a)
      if a>c do c := a → listed
  }
}
```

[5] This is not a proper extension as some concepts expressible in the original framework are no longer expressible. For example, partial matches or multiple verdicts. The main reason is that the original framework was defined in terms of *matching* and triggering advise whilst this framework is defined in terms of *correctness*.

The specification on the left is a variation of a property given in [44] and demonstrates existential quantification. It specifies the property that for every quadrant q there exists a satellite s such that every rover r in q has pinged s and received an acknowledgement i.e. there is a known single point of contact in that quadrant. The specification on the right is from [5] and specifies that bids on an item placed for auction should be strictly increasing. To support local state in a useful way it was necessary to introduce the notion of variables that do not take part in slicing (called *free* variables in this work).

Like RuleR, QEA has a four-valued semantics allowing for anticipatory results i.e. there are *false* and *true* verdicts if all extensions of a trace have the same verdict and *still-false* and *still-true* verdicts otherwise. An example where *false* may be returned is where a quantification is purely universal and slice enters a state from which no accepting state is reachable. Whilst the addition of local state and arbitrary actions and guards on transitions can theoretically make the expressiveness of QEA Turing-complete, overuse of such features can make QEA unreadable, arguably rendering the *usable* expressiveness almost regular. The automaton model means that specifications often capture low-level details. This can lead to less readable specifications than in, e.g., temporal logic [45] and a *plug-in* style approach as taken by JavaMOP may be beneficial in the future.

5.4 Nfer

Complex Event Processing (CEP) can be characterized as *event abstraction*, where a stream of low-level events are aggregated and transformed into higher-level events. CEP can be used for further analysis and/or human comprehension, e.g. through visualization. We here briefly describe nfer [56], in part influenced by our work on rule-based systems, and LogFire in particular. Consider the command example, where we monitor events such as Command(time,kind,name,nr), Distpatch(time,name,nr), and Success(time,name,nr). Assume further that an event Starvation indicates that a task on board the spacecraft is starved from executing. We now want to highlight the situation where a starvation warning is issued during a period where at the same time there is Earth communication activity and data-fetch (from the cameras) activity. The following nfer specification defines this scenario.

```
command :− Dispatch before Success
    where Dispatch.name = Success.name & Dispatch.nr = Success.nr
    map {name → Dispatch.name}
communication :− command where command.name = "COMM"
fetchdata :− command where command.name = "FETCH"
starvation :− Starvation during (communication intersect fetchdata)
```

The result of a applying an nfer specification to an event stream is a set of intervals, tuples of the form (η, t_1, t_2, m) consisting of a name η, a start time t_1, an end time t_2, and a map m holding data. The specification consists of four interval-generating rules, each of the form: name :− body (a rule name followed by a rule body). The semantics is similar to that of Prolog (hence the :− symbol): when

the body is true an interval is generated with that name. A difference from Prolog is that rule bodies contain temporal constraints. The first rule defines an interval describing the execution of a command as occurring between a command dispatch and a subsequent success where the command names and numbers match. The resulting command interval will also have an associated map that maps x to the command name. The next two rules named communication and fetchdata define the intervals where communication and data fetching commands are executed. The rule starvation captures the starvation occurring during the intersection of communication and data fetching. Other temporal operators (inspired by Allen temporal logic), include: **meet**, **coincide**, **start**, **finish**, and **overlap**. Rules can also access and explicitly reason about time values.

6 2003-2017 - Sound Predictive Runtime Analysis

An increasingly important class of runtime analysis algorithms are concerned with *predicting* anomalies in programs from *successful* observed executions. Two such early algorithms implemented in JPaX, one for predicting deadlocks and another for predicting data-races, were discussed in Section 2.1. Both of those algorithms are *unsound*, that is, they can and do report false positives. In contrast to static analysis, in predictive runtime analysis a sound algorithm is one which predicts only real errors, i.e., no false alarms. We discuss two categories of sound algorithms, one based on vector-clocks and another based on SMT solving.

6.1 Vector-Clock Based Algorithms: From JMPaX to jPredictor

A series of sound predictive runtime analysis algorithms and tools have been proposed for multi-threaded systems about a decade ago, based on *vector clocks* [33, 62] and on techniques proposed by the distributed systems debugging community, e.g., [76, 26, 19]. The main idea is to instrument the multi-threaded program to emit events timestamped by vector clocks, thus enabling the observer to extract a partial order reflecting the causal dependency on memory accesses. If any linearization of that inferred partial order leads to a violation of the desired property then an error is reported to the user, with the meaning that there are (likely different from the observed one, but definitely feasible) executions of the multithreaded program which violate the requirements.

Our first vector-clock-based predictive runtime verification tool was Java MultiPathExplorer (JMPaX) [70], briefly explained below. Consider the following multi-threaded program (in pseudocode) over shared variables x, y and z,

Initially: $x = -1;\ y = 0;\ z = 0;$

thread $T_1\{$	*thread* $T_2\{$
$x_{++};$	$z = x + 1;$
$y = x + 1;$	$x_{++};$
$\}$	$\}$

together with a desired property "if $(x > 0)$, then $(x = 0)$ has been true in the past, and since then $(y > z)$ was always false." Note that the shared variables may correspond to physical actions and thus violations of this property may result in potentially catastrophic system failures. This safety property can be formally specified using a past-time LTL formalism (similar to that used for JPaX in Section 2.1) but we keep the discussion informal here. A possible execution of the program can yield the sequence of states $(-1, 0, 0)$, $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 1)$, $(1, 1, 1)$, where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1$, $y = 0$, $z = 0$. This execution does *not* violate the desired property, so a normal runtime monitor would not report a violation. However, JMPaX' vector-clock based algorithm will infer, from the same execution above and without access to the actual code, that two other executions are possible (without false alarms) and that one of them in fact violates the property, namely $\{x = -1, y = 0, z = 0\}$, $\{x = 0\}$, $\{y = 1\}$, $\{z = 1\}$, $\{x = 1\}$, which corresponds to the sequence of states $(-1, 0, 0)$, $(0, 0, 0)$, $(0, 1, 0)$, $(0, 1, 1)$, $(1, 1, 1)$.

The vector-clock technique employed in JMPaX essentially implements a variant of Lamport's happens-before causality adapted to multi-threaded systems. Our colleagues have extended the technique in various ways, essentially demonstrating that increasingly more complex, yet more relaxed but still sound causal models can be considered, this way improving the predictive capability without reporting any false alarms; due to space constraints, we refer the reader to [72, 52] for a literature review. We have ourselves contributed by further extending the technique to consider various kinds of Java-like synchronization and communication primitives [69]. Finally, we noticed that in multi-threaded systems one can go beyond the usual happens-before causality [71]: a write event can be atomically grouped with all its corresponding subsequent read events, and that such groups of events can be permuted atomically; similarly, blocks of events in different threads protected by the same lock can be permuted atomically. As shown in [69, 71], these improvements led to significant increases in prediction capability without jeopardizing soundness. However, without taking into account information about the code of the program that generated the trace, that is without static analysis, we were not able to improve the vector-clock-based predictive runtime analysis algorithms any further.

jPredictor [25] was, to our knowledge, the first sound predictive runtime analysis system which combined static and dynamic analyses. Specifically, it implemented *sliced causality* [24], a happen-before causality drastically but soundly sliced by removing irrelevant causalities using semantic information about the program obtained with an a priori static analysis. Consider, e.g., a simple and common safety property for a shared resource, that any access should be authenticated, and consider the following buggy program executed as shown:

Main Thread {
```
  1. resource.authenticate();
  2. flag = true;
```
}

Task Thread {
```
  3. if(!flag) Thread.yield();
  4. resource.access();
```
}

The main thread authenticates and then the task thread uses the authenticated resource. They communicate via the `flag` variable. Synchronization is unnecessary, since only the main thread modifies `flag`. However, the developer makes a common mistake, using `if` instead of `while` in the task thread. Suppose now that we observed a successful run of the program, as shown above. Techniques based on traditional happen-before will not be able to find this bug, due to the causality induced by the write/read on `flag`. But since `resource.access()` is not controlled by `if`, sliced-causality techniques will correctly predict this error from the successful execution. When the bug is fixed replacing `if` with `while`, `resource.access()` is controlled by `while` (since it is a potentially non-terminating loop), so no violation is reported.

jPredictor is also implemented using vector clocks, but as discussed in [25], we were not able to obtain a faithful implementation. The vector-clock implementation was stronger than the sliced causality, thus maintaining soundness but potentially failing to report violations that were theoretically possible. In spite of the limitation, [25] experimentally showed that the combination of static and dynamic information cut, on average, about 50% of the dependencies, thus increasing the predictive capability of the technique exponentially. Unfortunately, probably due to the complex nature of resulting technique and its implementation, to our knowledge nobody continued to work in that direction. On the positive side, a new and appealing direction took shape, discussed below.

6.2 Maximal Causality and SMT-based Algorithms: RV-Predict

As mentioned above, the runtime verification community has developed increasingly more complex and more relaxed sound causal models of multithreaded system computations. A question naturally had arisen: is there an end to this quest? That is, is there a *maximal* causal model that we can extract from an observed trace, which cannot be surpassed? We answered this question positively for sequentially consistent systems [72, 67], essentially proposing a constructive causal model and showing the following: (1) all programs which can produce the observed execution can generate all traces in the model; and (2) for any trace t not in the model there exists a program generating the observed trace which cannot generate t. In other words, any sound and purely dynamic predictive runtime analysis technique can only detect a subset of the violations that the maximal causal model comprises (but albeit more efficiently). This result is foundationally very important, because on the one hand it draws a line in the sand w.r.t. how much sound predictive runtime analysis can go, and on the other hand it shows that the limit can be achieved.

Consider, for example, an execution of the program in Figure 2. The program contains a race between lines (`3`,`10`) that may cause an authentication failure of resource z at line `12`, which in consequence causes an error to occur when z is used at line `15`. Supposing the execution follows an order denoted by the line numbers, however, previous sound causal models cannot detect this race because line `3` causally-precedes line `10`, because the two lock regions contain conflicting accesses to y. While how to best use static analysis to further enhance

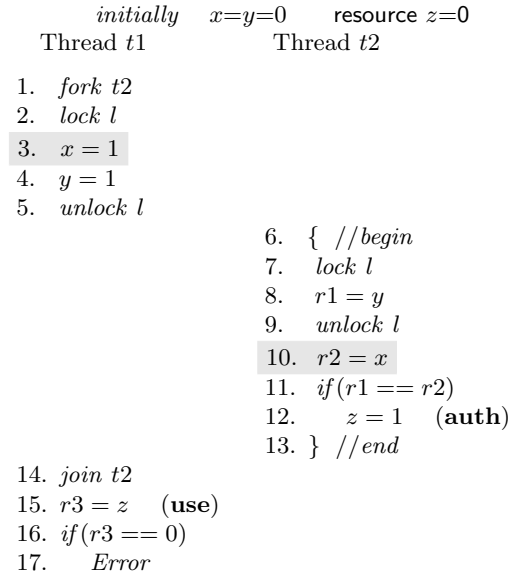

Fig. 2. An example program with a race (3,10).

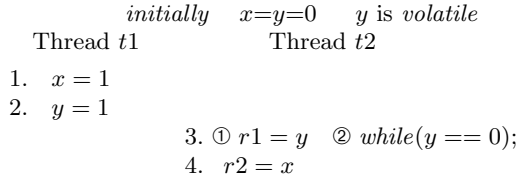

Fig. 3. The two cases ① and ② produce the same read/write trace. However, (1,4) is a race in case ① but not in case ②.

the maximal causal model is a valid question and worth pursuing, we found that the maximal causal model can already elegantly deal with information flow information if execution traces are enriched to also emit control-flow-changing (or branching) events [52]. Consider the scenario in Figure 3 where y is volatile and line 3 has two cases: ① $r1 = y$ and ② $while(y == 0)$. For case ①, (1,4) is a race on x; while for case ②, it is not, because line 4 is control-dependent on the while loop at line 3. However, without considering the control dependence between operations, the dynamic execution traces for these two cases are identical. But using the control flow information we can tell that, in case ①, line 4 is not control-dependent on line 3. In other words, regardless of what value line 3 reads, line 4 will always be executed. Therefore, we can safely drop the happens-before edge from line 2 to line 3, which enables detecting the race (1,4). Similarly, we are able to detect the race (3,10) in Figure 2 by dropping the happens-before edge from line 4 to line 8, because there is no control flow from line 8 to line 10 and hence no need to ensure line 8 should read value 1 (written by line 4).

A. Happens-before (Φ_{hb})	$O_1 < O_2 < \ldots < O_5 \wedge O_{14} < \ldots < O_{16}$ $O_6 < O_7 < \ldots < O_{13}$ $O_1 < O_6 \wedge O_{13} < O_{14}$
B. Locking (Φ_{lock})	$O_5 < O_7 \vee O_9 < O_2$
C. (3,10)Race (Φ_{race})	$O_{10} = O_3$

Fig. 4. Constraint modeling of the example execution in Figure 2.

The maximal causal model is more mathematically involved than the previous causal models, and it is still unknown whether it can be implemented using vector clocks. However, as Said etal. [67] first noticed, it is not very difficult to represent the maximal causal model as a mathematical formula. Specifically, we can associate to each event e in the trace one integer variable O_e, called its *order variable*, and then use the semantics of the various concurrent objects and control flow events to generate constraints over the order variables. For example, all the events emitted by the same thread must follow the same order as emitted (but can have other events interleaved), blocks protected by the same lock cannot overlap, and so on. Finally, one adds constraints for the property one is interested in; for a data-race, e.g., one says that the two involved events occur at the same time. Figure 4 shows the constraints for the execution in Figure 2.

The formula generated for a given trace therefore encodes all the ordering constraints that must be satisfied by any permutations of the events in the same trace in order to maintain soundness, as well as all the constraints that must be satisfied in order for the property of interest to be matched by the predicted trace. All is left now is to check the satisfiability of the resulting formula (e.g. with a SMT solver). If not satisfiable, then we can conclude that the observed execution trace has no evidence in it that the property is matched. If satisfiable, then a solution of it is a counter-example showing that there indeed exists a feasible execution of the system that match the property.

One might think that it is not practical to solve large formulae that can result from large traces. However, with some additional engineering and optimizations, the commercial RV-Predict tool (https://runtimeverification.com/predict) [52] has demonstrated not only that it can detect concurrency errors that no other predictive runtime analysis tools can, but also that it can do it at a relatively acceptable performance.

7 Reflections and Future Perspectives on RV

Logics The move from the early propositional temporal logics (such as JPaX) to parametric temporal logics (such as Eagle and MOP) was important, leading to an impressive community effort in researching logics and algorithms. The spectrum of specification logics has spanned many standard logics, such as automata, regular expressions, (future as well as past) linear temporal logics, context-free

grammars, variations of the μ-calculus, process algebras, stream processing, and rule-based systems. Most of these standard logics have had to be extended with first-order features to handle the parametric case [46]. In addition to the first-order trend, another trend has been the attempts to extend state machine notations with special states (such as the distinction between skip and next-states). Several attempts have been made in combining logics, specifically regular expressions and linear temporal logic, as in e.g. SALT [13]. These logics combine sequencing (adopted from regular expressions) with temporal operators. The LogScope language provided a formalism resembling a textual version of time lines and without explicit temporal operators such as *eventually*. The MOP system took a different view by providing a collection of different logics, such that each property is written in "the logic that fits" that property. An interesting logic framework is the modal μ-calculus, which e.g. is the basis for Eagle, where temporal properties and recursion can be combined with "named states". One particular promising aspect of Eagle was the support for user-defined temporal operators. Rule systems appear to be an interesting alternative to automata for the data parameterized case. However, traditional rule programs are in many cases not as readable as e.g. temporal logic. To improve this situation, they can be extended with syntactic sugar, e.g. state machine concepts, as done in RuleR. Rule systems can be powerful; for example, RuleR rules can take rules as arguments as a way of modeling context-free grammars. In RuleR, rule programs can be chained together with facts produced by one rule program becoming input to another rule program. This is related to stream processing. The idea of an event stream resulting in a set of facts/data can be viewed as Complex Event Processing (CEP), and is especially realized in the nfer system. This is an interesting avenue for future research. When formalizing a temporal property it can be useful to first to draw a time-line on a piece of paper, and then plot in events. This suggests that tool support for such a graphical time line approach might lower the barrier for writing temporal properties. Timelines have been studied in the context of model checking [75].

External versus Internal DSLs Whether to develop a DSL as external or internal is a non-trivial decision. An external DSL is usually cleaner and more directly tuned towards the immediate needs of the user. In addition, they are easier to process and therefore optimize for efficiency. However, the richer the DSL becomes (moving towards Turing-completeness) the harder the implementation effort becomes. An internal DSL can be very fast to implement and augment with new (even user-defined) operators, and can provide an expressiveness that would require a major effort to support in an external DSL. One also gains the advantage of IDEs etc. for the host language. However, some concepts may not be easily representable as an internal DSL. Also, a user will have to be a programmer in the host language. In this respect, some programming languages seem to be less of a barrier than others, e.g. Python is considered easy to learn.

A hypothesis is that monitoring logics used in practice will need to support very expressive expression languages to process data, such as strings and numbers that are part of the observed events. TraceContract is a shallow DSL in contrast

to LogFire, which is (mostly) a deep DSL. As a shallow DSL, TraceContract relies on Scala's type system. In contrast, for LogFire such a type system would have to be implemented from scratch. Also, in LogFire names have to be symbols or strings, which is somewhat annoying. LogScope was a compromise where the core DSL was external but with "holes" where one could write Python code, much like how parser generators such as `yacc` function. This was only possible due to Python's capabilities for evaluating a text string as a program (the `eval`-function), and would not, e.g. be possible in Java or Scala.

Programming Languages Temporal logic could become part of a programming language assertion language. This could be seen as part of a design-by contract approach also supporting pre/post conditions and class invariants. Libraries can come equipped with such temporal assertions verifying their correct use. The paper [20] in this volume discusses what to expect from future programming languages, and specifically likewise mentions support for "richer specifications" supported by stronger static and dynamic analysis. Adding such concepts to a programming language would be easier if the language came equipped with syntax extension/meta programming frameworks, a need we have often experienced in our work.

Aspect-oriented Programming Aspect-oriented programming has been a popular way of instrumenting Java programs for runtime verification. Although research in aspect-oriented programming seems to have slowed down, we do believe that the ideas of vertical (enriching pointcut language) and horizontal (stateful aspects) extensions of AOP are interesting, and should be part of a programming language's meta-programming environment. AOP is a natural host for RV. That is, rather than using AOP to instrument for RV, RV can be considered as a natural extension of AOP. Note, however, that not all RV solutions require such a close integration with a programming language; e.g. web service monitoring does not require this form of integration.

RV Oriented Requirements Engineering An intriguing thought is an approach to requirements engineering where at least *events* become part of the formal vocabulary, and where the implementation of the designed system is obliged to generate logs of such events, which can then be monitored. Logging (and monitoring) should become part of programming larger systems.

Algorithms Concerning monitoring algorithms, the slicing-based algorithms, as found in Tracematches, MOP, and QEA, have so far shown to be the most efficient, initially at the cost of limited expressiveness, but in QEA extended to allow for improved expressiveness. Experiments such as the use of the Rete algorithm in LogFire, or the use of SMT [29] in MMT (Monitoring Modulo Theories) have not shown the same degree of performance. We still think, however, that new algorithms for parametric monitoring are of interest, especially since the original limitations wrt. expressiveness can be considered a major issue. In [42] we e.g. experiment with the use of BDDs for monitoring first-order past time temporal logic, with interesting performance results.

Predictive Monitoring The earliest examples of predictive algorithms for deadlock and data race detection from Compaq were very promising, and showed

to be exceptionally effective in practice. Later results using SMT have shown tremendous potential.

Beyond Boolean Specification-based runtime verification approaches tend to be Boolean valued algorithms: determining whether a sequence of events satisfies a temporally oriented specification. That is, $M(\sigma) \in \mathbb{B}$ (or some simple extension $\mathbb{B}^+$ of $\mathbb{B}$). However, as stated in Section 1, runtime verification in its generality can be considered as computing any kind of value, $M(\sigma) \in D$, for any domain D. We already encountered the nfer system which computes intervals (D is the set of intervals). In [35], a very early approach to computing values from a trace driven by temporal formulas is described. In other approaches, the result is a probability for a property to be satisfied, as in [77] (see discussion below). In statistical model checking [58], see also [60] in this volume, a stochastic system is executed multiple times, monitoring each execution against a temporal formula, computing either the probability that the system satisfies a formula (quantitative SMC), or determining whether the probability is greater than or equal to a certain treshold (qualitative SMC).

Specification Mining and Inference We consider the 'mining' or 'learning' of specifications from traces to be a very promising field. Here we consider some work in this area (including our own e.g. [64, 77, 59]) but do not make an attempt to be complete. There exist general introductions to the topic [2, 61, 28]. In [77], an approach named *Runtime Verification with State Estimation* (RVSE) is described, which uses learning to estimate the probability that a trace with missing events (gaps in the trace) satisfies a given temporal property. This idea can, for example, be applied when monitoring overhead is reduced by sampling. The strategy is to learn the nominal behavior (without gaps) of the system as a Hidden Markov Model (HMM), and the later use this model to "fill in" sampling-induced gaps in an observed trace. Two approaches have attempted to use parametric trace slicing to learn parametric specifications. In [59], a probabilistic automata learning algorithm was applied to trace slices to build a hypothesis specification which was then heuristically refined. In [64] many predefined patterns were checked against trace slices and then combined to form ranked hypothesis specifications. Further work in both directions, and in specification mining in general, seems important to the field of runtime verification as the lack of specifications is sometimes cited as a barrier to application of RV. The above work was *passive* in the sense that it took as an input a given set of traces. Another promising direction is the area of *active* automata learning where queries may be given to build a (in some contexts) complete specification of behavior. One of the more advanced instances of this approach [53] is the learning of *register automata* – an extension of finite automata where data values may be communicated, stored and manipulated. In this sense, this work corresponds to the parametric approaches mentioned above. Additionally, an approach is described in the paper [51] in this volume for combining black-box (no access to code) and white-box (access to code) techniques. These active learning techniques are implemented in the well-known LearnLib tool [55]. Recent work [54] has adapted the framework to handle the long traces encountered in RV.

Trace Visualization Execution trace visualization is a subject that in our opinion has promising potential, although our own work in this direction is limited to [4] and nfer (where the intent is to visualize event hierarchies). The advantage of visualization is that it can provide a free-of-charge abstract view of the trace, from which a user potentially may be able to conclude properties about the program, or at least the execution, without having to explicitly formulate these properties. We can distinguish between two forms of trace visualization: *still visualization*, where all events are visualized in one view, and *animated visualization*. In [4], an extension of UML sequence diagrams with symbols is described for representing still visualizations of the execution of concurrent programs. There appears to be a relationship between still visualization and automated specification mining. For example, a state machine learned from several runs can be regarded as a still visualization, as well as a specification of its behavior during those runs.

Combining Static and Dynamic Analysis Full verification is of course preferred over partial verification performed by a monitor. The combination of static and dynamic verification can provide the best of both worlds: prove as much as is feasible and verify the remaining proof obligations during runtime.

Runtime Enforcement and Fault Protection In runtime enforcement [31], one uses a monitor as a filter in front of a system, the target, receiving events from another system, the source. In this preventive approach, only events satisfying the property defined by the monitor will be let through to the target. In fault-protection strategies, the goal is to recover the system once it has failed; see e.g. [11] where this is called *adaptive runtime verification*. Here, two versions of the program being monitored exist: the complex version (running by default) and the simple version, and in case of a property violation the simple version overtakes the complex version. The general problem of how to recover from a bad program state is interesting and quite challenging. The ultimate solution to this problem can be found in planning and scheduling systems, where a planner creates a plan (straight-line program) to execute for a limited time period, an executive executes the plan, and a monitor monitors the execution. Upon failure detected by the monitor, a new plan (program) is generated online.

Summary Searching for the most efficient monitoring algorithms, balanced with expressiveness of logics, is an ongoing research topic. The field has studied and produced an interesting set of temporal logics, that differ from logics produced by the field of e.g. model checking, in part due to the different application scenario, such as focus on single traces with data carrying events. This includes the distinction between external and internal DSLs, AOP, and logics for computing data (beyond the Boolean domain) from traces. Avoiding writing specifications, as pursued in specification mining and predictive monitoring, is an interesting line of research with a lot of potential. The integration of static and dynamic analysis is another important line of research, that is in its infancy as well. Finally, it would be interesting to pursue an integration of temporal logic in programming languages as part of the assertion language.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
2. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.
3. C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4), 2004.
4. C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 541–546, July 2007.
5. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata - towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
7. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
8. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
9. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. *Journal of Logic and Computation*, 20(3):675–706, 2010.
10. E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, and G. Reger. An introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–23. Springer, 2018.
11. E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. In *Proc. of RV 2012, the 12th International Conference on Runtime Verification*, volume 7687 of *LNCS*, pages 168–182. Springer, 2012.
12. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 126–138, Vancouver, Canada, 2007. Springer.
13. A. Bauer, M. Leucker, and J. Streit. SALT – structured assertion language for temporal logic. In *Proc. of the 8th International Conference on Formal Methods and Software Engineering (ICFEM'06)*, volume 4260 of *LNCS*, pages 757–775. Springer, 2006.
14. M. Bennett, R. Borgen, K. Havelund, M. Ingham, and D. Wagner. Prototyping a domain-specific language for monitor and control systems. *Journal of Aerospace Computing, Information, and Communication*, 7(11):338–364, 2010.
15. S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *LNCS*, pages 208–223. Springer, 2006.
16. E. Bodden. MOPBox: A library approach to runtime verification. In *Proc. of RV 2011, the 11th International Conference on Runtime Verification, San Francisco,*

USA, September 27-30, 2011. Proceedings, volume 7186 of *LNCS*, pages 365–369. Springer, 2011.
17. E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Trans. Softw. Eng.*, 36(4):509–527, July 2010.
18. G. Candea and P. Godefroid. Automated software test generation: Theory and practice. In *Issue Number 10000 of Lecture Notes in Computer Science*, volume 10000 of *LNCS*. Springer, 2018. In this volume.
19. C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
20. R. Chatley, A. Donaldson, and A. Mycroft. The next 7000 programming languages. In *Issue Number 10000 of Lecture Notes in Computer Science*, volume 10000 of *LNCS*. Springer, 2018. In this volume.
21. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 357 – 373. Springer-Verlag, 2004.
22. F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proc. of the 3rd Int. Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Elec. Notes Theo. Comput. Sci.*, pages 108 – 127. Elsevier Science Inc., 2003.
23. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM, ACM SIGPLAN Notices, 2007.
24. F. Chen and G. Rosu. Parametric and Sliced Causality. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 240 – 253. Springer, 2007.
25. F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE*, 2008.
26. R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.
27. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
28. C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.
29. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 341–356. Springer, 2014.
30. D. Drusinsky. The temporal rover and the ATG rover. In *Proc. of the 7th International SPIN Workshop on Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
31. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Trans.*, 14(3):349–382, 2012.
32. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. Peled, and G. Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
33. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194. ACM, 1988.

34. R. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Foundations of Aspect-Oriented Languages (FOAL'02), Enschede, The Netherlands*, April 2002.
35. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.
36. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN 2000, Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
37. K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
38. K. Havelund. Runtime verification of C programs. In *Proc. of the 1st TestCom/-FATES conference*, volume 5047 of *LNCS*, Tokyo, Japan, 2008. Springer.
39. K. Havelund. Data automata in Scala. In *Proc. of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE'14)*. IEEE Computer Society, 2014.
40. K. Havelund. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Trans.*, 17(2):143–170, 2015.
41. K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, 2008.
42. K. Havelund, D. A. Peled, and D. Ulus. First order temporal logic monitoring with BDDs. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 116–123. IEEE, 2017.
43. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.
44. K. Havelund and G. Reger. Specification of parametric monitors - quantified event automata versus rule systems. In *Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, pages 151–189. Springer Fachmedien Wiesbaden, September 2015.
45. K. Havelund and G. Reger. Runtime verification logics - a language design perspective. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, Aalborg University, 19-20 August 2017*, volume 10460 of *LNCS*, pages 310–338. Springer, 2017.
46. K. Havelund, G. Reger, D. Thoma, and E. Zălinescu. Monitoring events that carry data. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification, Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 61–102. Springer, 2018.
47. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.
48. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 135–143, 2001.
49. K. Havelund and W. Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.
50. K. Havelund and E. V. Wyk. Aspect-oriented monitoring of C programs. In *The Sixth IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, Pasadena, CA, May 17-18*, 2008.

51. F. Howar, B. Jonsson, and F. Vaandrager. Combining black-box and white-box techniques for learning register automata. In *Issue Number 10000 of Lecture Notes in Computer Science*, volume 10000 of *LNCS*. Springer, 2018. In this volume.
52. J. Huang, P. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*, pages 337–348. ACM, June 2014.
53. M. Isberner, F. Howar, and B. Steffen. Learning register automata: From languages to program structures. *Mach. Learn.*, 96(1-2):65–98, July 2014.
54. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Proc. of RV 2014, the 14th International Conference on Runtime Verification*, volume 8734 of *LNCS*, pages 307–322. Springer, 2014.
55. M. Isberner, F. Howar, and B. Steffen. The open-source learnlib. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *LNCS*, pages 487–495. Springer, 2015.
56. S. Kauffman, K. Havelund, and R. Joshi. nfer – a notation and system for inferring event stream abstractions. In *Proc. of RV 2016, the 16th International Conference on Runtime Verification, Madrid, Spain, September 23–30, 2016, Proceedings*, volume 10012 of *LNCS*, pages 235–250. Springer, 2016.
57. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
58. K. G. Larsen and A. Legay. Statistical model checking: Past, present, and future. In T. Margaria and B. Steffen, editors, *7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2016, Corfu, Greece, October 10-14*, volume 9953 of *LNCS*, pages 3–15. Springer, 2016.
59. C. Lee, F. Chen, and G. Rosu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 591–600, 2011.
60. A. Legay, A. Lukina, L. M. Traonouez, J. Yang, S. A. Smolka, and R. Grosu. Statistical model checking. In *Issue Number 10000 of Lecture Notes in Computer Science*, volume 10000 of *LNCS*. Springer, 2018. In this volume.
61. D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining software specifications: methodologies and applications*. CRC Press, 2011.
62. F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier science, 1989.
63. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 249–289, 2011.
64. G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 658–663, Nov 2013.
65. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 596–610. Springer, 2015.

66. G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.
67. M. Said, C. Wang, Z. Yang, and K. A. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods'11*, volume 6617 of *LNCS*, pages 313–327, 2011.
68. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
69. K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 123–138. Springer, 2002.
70. K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proc. of ESEC/FSE'03: European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, Helsinki, Finland, September 2003.
71. K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS'05*, volume 3535 of *LNCS*, pages 211–226, 2005.
72. T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *Proc. of RV 2012, the 12th International Conference on Runtime Verification, Istanbul, Turkey*, volume 7687 of *LNCS*, pages 136–150. Springer-Verlag, 2012.
73. J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. InterAspect: aspect-oriented instrumentation with GCC. *Formal Methods in System Design*, 41(3):295–320, 2012.
74. D. R. Smith and K. Havelund. Toward Automated Enforcement of Error-Handling Policies. Technical Report number: TR-KT-0508, Kestrel Technology LLC, August 2005.
75. M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *21st IEEE International Requirements Engineering Conference (RE), Toronto, Canada*, August 2001.
76. S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.
77. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proc. of RV 2011, the 11th International Conference on Runtime Verification, San Francisco, CA, USA*, volume 7186 of *LNCS*, pages 193–207. Springer-Verlag, 2011.
78. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *Elec. Notes Theo. Comput. Sci.*, pages 109–124. Elsevier Science Inc., 2006.
79. V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *Elec. Notes Theo. Comput. Sci.*, pages 201–216. Elsevier Science Inc., 2005.
80. R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.