Kalpa Publications in Computing
Volume 3, 2017, Pages 123–137
RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools

On the Evaluation and Comparison of Runtime Verification Tools for Hardware and Cyber-Physical Systems*

Kristin Yvonne Rozier[1]

Iowa State University, Ames, Iowa, U.S.A.
KYRozier@iastate.edu

Abstract

The need for runtime verification (RV), and tools that enable RV in practice, is widely recognized. Systems that need to operate autonomously necessitate on-board RV technologies, from Mars rovers that need to sustain operation despite delayed communication from operators on Earth, to Unmanned Aerial Systems (UAS) that must fly without a human on-board, to robots operating in dynamic or hazardous environments that must take care to preserve both themselves and their surroundings. Enabling all forms of autonomy, from tele-operation to automated control to decision-making to learning, requires some ability for the autonomous system to reason about itself. The broader class of safety-critical systems require means of runtime self-checking to ensure their critical functions have not degraded during use.

Runtime verification addresses a vital need for self-referential reasoning and system health management, but there is not currently a generalized approach that answers the lower-level questions. What are the inputs to RV? What are the outputs? What level(s) of the system do we need RV tools to verify, from bits and sensor signals to high-level architectures, and at what temporal frequency? How do we know our runtime verdicts are correct? How do the answers to these questions change for software, hardware, or cyber-physical systems (CPS)? How do we benchmark RV tools to assess their (comparative) suitability for particular platforms? The goal of this position paper is to fuel the discussion of ways to improve how we evaluate and compare tools for runtime verification, particularly for cyber-physical systems.

1 Introduction

Runtime Verification (RV), or the formal analysis of a system requirement for a single run of the system (the current run), in real-time, is vital to the future of safety-critical, and particularly autonomous systems. Whether we consider autonomy to be as simple as operating independently, such as an Unmanned Aerial System (UAS) without a human on-board that depends on a remote pilot, to being entirely free from external control, such as the same UAS with only on-board control, we need some way for the system to reason about itself. The UAS must be

*Work supported in part by NASA Autonomy Operating System (AOS) 8042-018286_NASA NNX14AN61A, NASA ECF NNX16AR57G, and NSF CAREER Award CNS-1552934.

able to notice at least the kinds of faults that would be noticed by a human pilot, were there one on-board. This type of self-referential reasoning becomes even more important when we consider more advanced definitions of autonomy, such as decision-making, e.g., the ability to freely choose from a set of actions while taking into account the current system status, or learning, which we can broadly define as any nondeterministic behavior. In all of these cases, RV is a fundamental component of System Health Management (SHM), or the ability for an autonomous system to evaluate its own status with regard to operational and mission requirements.

Due to their unique ability to accurately pinpoint anomalies not found by other verification techniques, particularly those of a complex, temporal nature, formal methods have greatly impacted the design and development processes of real-life, full-scale, safety-critical systems. For example, in the aerospace industry, design-time model checking of temporal logic formulas [20] has increased the robustness of the Small Aircraft Transportation System (SATS) [17], verified the Traffic Alert and Collision Avoidance System (TCAS) flying on-board commercial aircraft [2], ensured internal aircraft modes followed the A-7E aircraft software requirements [25], robustified Boeing's AIR6110 wheel braking system [7], analyzed the Mars Science Laboratory's flight software [14], and changed NASA's design for the NextGen automated air traffic control system [12, 16, 26]. Our challenge now is to carry this success from design-time to runtime.

The Competition on Runtime Verification (CRV) [4] marked an important step forward in evaluation of RV *for software*; the core idea of software RV is to instrument a program to emit events during its execution, which are then processed by a monitor. Like CRV, we consider specification-based trace analysis, where execution traces are verified against formal specifications written in formal logical systems.

We need to recognize that (a) hardware and cyber-physical (hardware-software combination) systems cannot be instrumented in the same way that software can be instrumented; (b) many systems (hardware, software, or cyber-physical) cannot be instrumented at all. For example, autopilot software represents a safety-critical component of an aircraft that requires runtime verification yet all autopilots running on commercial aircraft fall somewhere on the spectrum from unable to be instrumented due to flight certification procedures, to closed-source, to ITAR[1] to some variation of classified. Runtime verification for such systems needs to avoid instrumentation and operate on external observations. **Therefore the focus of this paper is on verification during system runtime that does not require software instrumentation. We also consider comparative evaluation in terms of correctness, timing, and overhead differently from CRV**, in the context of hardware and cyber-physical systems.

We address, in part, each of the six questions itemized by the International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES).[2] We address these important questions from the domain of safety-critical aerospace systems, focusing on hardware and cyber-physical systems.

1. **What should a RV benchmark look like?**
2. **Can we have a common specification language for RV? If so, what should it look like?**
3. **Is execution time the most important performance criteria? What might be more important?**
4. **How can we evaluate hardware monitoring tools?**
5. **What are we doing wrong in evaluation? Can we fix this?**

[1] `https://www.pmddtc.state.gov/regulations_laws/itar.html`
[2] http://rv2017.cs.manchester.ac.uk/rv-cubes/

6. **What can be borrowed from other communities?**

We frame our discussion in terms of the features unique to verification *during system runtime*, also called *online verification.* We distinguish this from offline verification, or the post-mission verification of recorded single-run executions.

1.1 Defining Features of RV

Before we can address these six questions, we must first ask the zeroth question: what are we doing RV for? It is essential to keep in mind the point of doing RV in the first place, and what we expect the outcome(s) to be. We argue that RV has some unique defining features that pose particular challenges for benchmarks and evaluation.

Inputs. RV must overate over *continuing streams of discrete inputs at regular temporal intervals.* Unlike design-time verification where we may be able to directly reason about continuous properties of the physical system, RV algorithms must reason about all aspects of the system, be they inherently discrete or continuous, using discrete signals. This is because software values are inherently discrete, hardware logic is also discrete, and physical properties of the system that are inherently continuous can only be measured by discretized sensor signals with some well-defined maximum frequency.

Tools and Implementations. RV has the unique property that any algorithm for RV *must be able to be implemented as a tool that runs in real time* **on** *the actual system being verified.* We note that the word *on* in this sentence is interpreted rather loosely: RV for an aircraft, for example, can be performed entirely on-board the aircraft, entirely on some ground system that is continuously receiving values from the aircraft, or via some distributed combination of these two. Similarly, *in real time* requires a precise definition. Unlike design-time verification, where any execution platform that completes the verification run within machine timeout (usually 24 hours or more) can be helpful, RV algorithms *must be implementable as tools* that run both *in real time* and *on the system being verified.* Importantly, this places limitations on the execution platform; for example, a typical supercomputer is not an appropriate test platform for an RV tool because an actual RV tool would never wait in a queue (perhaps for days or weeks), execute on the next node available, and have its results sent back all at once. RV tools must be executed (and therefore benchmarked) on real-time execution platforms that may be constrained by the operational envelope of the system under test. RV tools that run on-board aircraft, for example, are subject to different constraints on their execution platforms depending on the particular aircraft system they aim to verify. Such constraints include power, weight, timing, size, cost, bandwidth, and interface limitations. We argue that RV tools also require *usability* sufficient to enable their installation, configuration, and use by system operators on realistic (embedded) platforms.

Outputs. RV tools must produce *continuing streams of discrete outputs (verdicts) in real time, in a format that can be utilized by the system being verified.* This criteria is arguably the most ill-defined criteria for RV, and therefore the most in need of standardization. Unlike design-time verification tools, RV cannot produce a single output; the output must be a stream corresponding to the stream(s) of inputs as the system runs. The *real time* constraint considers both execution time and delays required to amass all of the data required to complete evaluation of a specification for a particular time window: how soon can we know the verdict given the input data, and how fast can we perform any computations necessary to compute it. We must decide whether the output stream should follow some regular temporal frequency or report verdicts as soon as they are computed, even if this results in outputs corresponding to different inputs being reported out-of-order or at irregular intervals. Finally, there is the amorphous criteria that the output should be in a format that can be utilized by the actual system we are currently verifying, which is a concept unique to RV. For an aircraft, for example, being able to utilize the RV outputs could mean that they can be used by another system (like a planner or autopilot) on-board the craft, that they can be used by an on-ground system (like an air traffic manager), or that they can be stored for post-accident diagnosis (like in a cockpit black box).

Linear Time Specifications. RV analyzes a single execution of the system; for RV time is inherently linear. Alternative notions of specifying time, such as branching time, do not apply during runtime. For design-time reasoning, such as model checking, a temporal logic specification φ is often a Linear Temporal Logic (LTL) formula. For RV, we often define φ to be either LTL or a finitely bounded variation of LTL, such as MTL [1], STL [15], or LTLf [8]. However, the system under test may dictate that φ describe probabilistic, or cyber-physical requirement, capturing, for example, partial differential equations tied to the physical properties of the system. For the purpose of generality, in this paper we presume MTL specifications. Systems in our application domain are usually bounded to a certain finite mission time. For example, a UAS has a limited air-time, depending on the available battery capacity and predefined waypoints.

Similarly to design-time checking, we can can define runtime verification formally over system computations. We define a computation π to be a sequence of system states, corresponding to the behavior of the system under test starting and then running until the end of the mission, which is usually finite for systems targeted by runtime verification. (While some safety-critical systems employing runtime verification run indefinitely, such as an automated air traffic control system, most execute finite missions, such as an aircraft or spacecraft.)

Let φ be a temporal formula specifying a requirement of the system that must hold for the system to be considered "correct" or "healthy." We satisfy such formulas over *computations*, which are functions that assign truth values to the variables of φ at each time instant [11].

Definition 1 (Computation)**.** *We interpret LTL formulas over computations of the form $\pi : \omega \rightarrow 2^{Prop}$, where ω is used in the standard way to denote the set of non-negative integers; for MTL formulas we replace ω with the finite mission time for the system under test. We write $\pi, i \vDash \varphi$ to designate that computation π at time instant $i \in \omega$ satisfies formula φ.*

For mission time m, the RV question can be formally defined as, $\forall i : 0 \leq i \leq m, \pi, i \vDash \varphi$. We answer this question in terms of an execution sequence. Given an input stream of time-stamped events, collected incrementally from the analyzed system, and an MTL specification φ, we define the outputs of RV in terms of *execution sequences*.

Definition 2 (Execution Sequence [18])**.** *An execution sequence for an MTL formula φ, denoted by $\langle T_\varphi \rangle$, is a sequence of tuples $T_\varphi = (v, \tau_e)$ where $\tau_e \in \mathbb{N}_0$ is a time stamp and $v \in \{\textbf{\textit{true}}, \textbf{\textit{false}}\}$ is a verdict.*

We use a superscript integer to access a particular element in $\langle T_\varphi \rangle$, e.g., $\langle T_\varphi^0 \rangle$ is the first element in execution sequence $\langle T_\varphi \rangle$. We write $T_\varphi.\tau_e$ to access τ_e and $T_\varphi.v$ to access v of such an element. We say T_φ holds if $T_\varphi.v$ is **true** and T_φ does not hold if $T_\varphi.v$ is **false**. For a given execution sequence $\langle T_\varphi \rangle = \langle T_\varphi^0 \rangle, \langle T_\varphi^1 \rangle, \langle T_\varphi^2 \rangle, \langle T_\varphi^3 \rangle, \ldots$, the tuple accessed by $\langle T_\varphi^i \rangle$ corresponds to a section of an execution e as follows: for all times $n \in [\langle T_\varphi^{i-1} \rangle.\tau_e + 1, \langle T_\varphi^i \rangle.\tau_e]$, $e^n \vDash \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **true** and $e^n \nvDash \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **false**.

An RV Benchmark is then an execution sequence for some input specification language (e.g., MTL), possibly paired with a *purpose*, an English description of what the benchmark aims to assess.

The rest of this paper is organized around the central questions of RV-CUBES. Section 2 breaks down the structure of an RV benchmark and puts forward ideas for standardizing specification languages for the RV competition. Evaluation criteria, including a discussion on the question of execution time make up Section 3. Section 4 briefly highlights ideas specific to evaluating hardware monitoring tools. We consider what best-practices and lessons-learned we can borrow from other communities in Section 5. Section 6 concludes and offers an outlook on the future of the RV competition.

2 Structure of a CPS RV Benchmark

In its most basic form, an RV benchmark *for hardware or cyber-physical systems*[3] needs three components, per Definition 2 (pictured in Figure 1):

1. a discrete-time **input stream** for each system variable we wish to reason about
2. a **requirement** to evaluate over that set of input streams
3. an **output stream** of verdicts: for each time step, did the requirement hold?

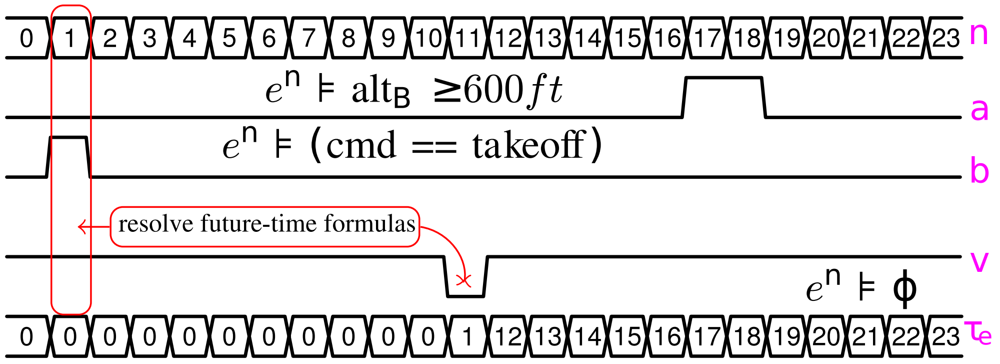

Figure 1: We exemplify an RV benchmark using Definition 2. For all times n, $0 \leq n \leq m$ for finite mission time m, the **input streams** to a runtime observer for future-time MTL formula φ specify the values of the variables in φ. For example, the input stream for Boolean variable a is an execution sequence $e^n \models alt_B \geq 600\ ft$ (the barometric altimeter reading is greater than 600 ft [18]). The input stream for b is an execution sequence $e^n \models$ (cmd == takeoff) (command to takeoff was received). The **requirement** is a future-time MTL formula φ over variables a and b. The **output steam**, $\langle T_\varphi \rangle$, is a sequence of tuples $T_\varphi = (v, \tau_e)$ where $\tau_e \in \mathbb{N}_0$ is a time stamp and $v \in \{\textbf{true}, \textbf{false}\}$ is a verdict. When the requirement is specified in future-time logic the output stream may need to aggregate values. Let t_i be the first time step where there is sufficient information to determine a correct verdict for time i here the output stream is $(\textbf{true}, 0)$, where $t_0 = 0$, $(\textbf{false}, 1)$, where $t_1 = 11$, $(\textbf{true}, 12)$ (indicating φ holds at times $2, \ldots, 12$), where $t_2 \ldots t_{12} = 12$, ...

[3]Note that these benchmarks for *online* monitoring of hardware or cyber-physical systems most closely resemble in format the benchmarks *offline* monitoring of software defined in [4] since online monitoring of software requires instrumented program source code.

For each of these three basic components, several details need to be standardized. One option is to create classes of benchmarks, each with a different combination of choices from the following lists.

Input Stream(s). Each system variable that is utilized needs to be supplied as an input stream simulating a run of the target system being verified over the length of a mission. For these input streams, we need to be standardize the *frequency* and *format*.

Frequency, or the space between "time steps" in the temporal logic notion of arbitrary units needs to be tied to some notion of real time as in "once per tick of the system clock" or other common frequency, given in Hz. For real-life systems, there may be multiple different frequencies across the set of input streams, realistically representing sensors that sample at different rates, and a single input stream may change frequencies, like the many spacecraft subsystems that take increasingly frequent measurements as touch-down approaches.

The **format** of an input stream can be either Boolean, or non-Boolean paired with Boolean testers to create Boolean streams to populate the propositions of an MTL specification. Non-Boolean data may be filtered or may require filtering within the RV tool; for fair benchmarking purposes, which filter to use should be given to enable benchmarking the inclusion of the *same* filter in the RV tool as part of the analysis. For example, a requirement might be "$(a < 5) \rightarrow (b > 20)$ where a and b are given as raw sensor streams; the RV tool would then be expected to execute the Boolean testers $(a < 5)$ and $(b > 20)$ to evaluate the requirement. We must decide: where we get the inputs from? Input streams can be randomly generated, come from software benchmarks, or come from Boolean testers over sensor input signals from real system executions. For example, real flight data from NASA UAS test flights is available online.[4]

Requirements. Benchmark requirements may be individually grouped with the set of input streams they reason over, or organized as a set of requirements over a shared set of input streams. Either way, requirements should be classified according to their *specification language* (temporal logic, probabilistic, or cyber-physical; see Section 2.1), and *performance constraints*, such as limits on the resources (e.g., memory, overhead) that may be used or bounds on the time verdicts may be delivered.

Another important question is: where do we get the requirements from? The answer relates to what format the requirements are given in: English or formal semantics? If we pull requirements from requirements documents, such as operational concepts for aerospace systems, then we could include in the RV competition the act of converting them to an appropriate formalism, similar to the VerifyThis competition,[5]. We can also create standardized benchmarks inspired by real life, and extracted from requirements for previous case studies. We can also populate the benchmark suite by generating requirements manually, though generating realistic requirements for a given system is an active area of research [19].

Output Streams. We expect output streams to be tuples, at least containing a verdict paired with the time for which that verdict holds. Following Definition 2, for example, $\langle T_\varphi \rangle = ((\mathbf{false}, 0), (\mathbf{false}, 1), (\mathbf{false}, 2), (\mathbf{true}, 3), \ldots, (\mathbf{true}, 17), (\mathbf{true}, 18))$ describes $e^n \models \varphi$ sampled over $n \in [0, 18]$ [18]. In addition to Boolean output streams, we may also evaluate over three-valued (true, false, maybe) or otherwise fuzzy output streams; in the case $\langle T^i_\varphi \rangle$ is **maybe**, neither $e^n \models \varphi$ nor $e^n \not\models \varphi$ is defined. Alternative formats include probabilistic, e.g., giving the probability that requirement holds; tuples, e.g., grouping multiple outputs of different formats;

[4]`http://www.usgs.gov/blogs/surprisevalley/`

[5]`http://www.verifythis.org/`

or measured-distance outputs, e.g., specifying how much of the requirement holds or how far the system status is away from satisfying the requirement in terms of time or difference of measures such as 200 feet away from the 400 foot altitude requirement.

The question of how we generate the output streams is the most difficult because it involves solving the RV problem in the first place; it is not easy to check or verify that the outputs are correct. We can approximate correctness before we have a known, canonically correct output for a benchmark, by taking a survey of a large number of solutions and figuring that most of them are correct, or by creating the benchmark by starting with the output and moving backwards to get a requirement and input that generate that output correctly. One major point to consider is that for some combinations of inputs and requirements it may not be possible within the real-time parameters to determine the output correctly due to the computational complexity of the problem. In this case, a "correct" output would be a timeout or statement that there will be no output. It is very important to distinguish this from something that resembles an output, and therefore could be used incorrectly by the system.

Since RV tools are chiefly used in critical applications, it is extremely important that we do **not rank tools that produce incorrect outputs**. Even a few incorrect outputs, that are not decline-to-answer or timeouts, could have seriously detrimental effects if they were used to modify the behavior of real safety-critical systems during runtime. In the case a tool produces any incorrect output, the test results should be returned to the tool authors for debugging, but not included in the competition. Focusing on correctness also motivates making the benchmark suite publicly available for tool debugging before the competition is held.

2.1 Specification Languages for RV

Ideally, the RV competition should standardize around the minimum number of specification languages required to express system requirements that need to be verified at runtime. We define the specification language to be the language of φ in the RV question of, for mission time m, $\forall i : 0 \leq i \leq m, \pi, i \models \varphi$. Wherever possible, translators should be used for all languages with overlapping expressibility to minimize the number of competition languages. (This is also a lesson learned from other verification competitions, such as HWMCC.[6]) Given that RV is used for Boolean, temporal, and probabilistic analysis and that many of the specification languages parsed by current RV tools are expressively incomparable, this is a challenge! However, we argue that a reasonable cyber-physical RV competition could be constructed from temporal logic specifications with the possibility of future tracks expanding the specification languages.

Temporal Logic. The same temporal logic specifications could be used for the runtime verification of both hardware and software. Ideally, the RV competition would choose just one temporal logic for all benchmarks and then offer open-source translators from all other logics into that one, even if that means choosing a less-expressive common logic. MTL is a reasonable candidate for the standard benchmark logic; for example, LTL specifications can be translated via the following definition.

Definition 3 (Mission-Time LTL [18]). *For a given LTL formula ξ and a mission time $t_m \in \mathbb{N}_0$, we denote by ξ_m the mission-time bounded equivalent of ξ, where ξ_m is obtained by replacing every $\Box\varphi$, $\Diamond\varphi$, and $\varphi\,\mathcal{U}\,\psi$ operator in ξ by the $\Box_J\,\varphi$, $\Diamond_J\varphi$, and $\varphi\,\mathcal{U}_J\,\psi$ operators of MTL, where the duration $J = [0, t_m]$ for mission time bound t_m.*

[6]`http://fmv.jku.at/hwmcc`

Probabilistic. RV tools may realistically be tasked with answering questions like "what is the most likely status of the fluxgate magnetometer," given that the fluxgate magnetometer can possibly suffer any one of five faults or be healthy [13]. One of the roles RV-CUBES can play in the future is to help define and standardize the format for such specifications, such as a common database. There could be a table for each requirement and the table could have a row for each status, e.g., each type of fault or 'healthy,' and then columns of symptoms. Symptoms could be assignments to variables. Then an RV tool should detect, given streams from all of the variables, the most likely status and the probability of this status.

Cyber-Physical. Cyber-physical specifications combine a cyber (temporal logic formula) specification with a physical (partial differential equation) specification to describe integrated hardware-software requirements. The temporal logic components of these specifications may be able to serve double-duty as benchmarks by themselves. Such specifications may be mined, e.g., from robotics verification case studies.

3 Evaluation Criteria

Correctness needs to be the primary evaluation criteria. RV tools are added to systems to check that the current run satisfies mission-critical requirements; they are expected to sound an alarm as early as possible if not, in order to enable mitigation actions. The real-time nature of RV does not allow for double-checking the results and demands immediate actions when a critical violation is reported. The entire point of RV is to mitigate the faults of other verified systems; the RV system itself needs to be very careful not to also generate faults. The impact of incorrect answers is arguably more critical for RV than for any other category of formal verification.

Importantly from a benchmarking perspective, correctness is also very challenging. The RV question for temporal logic specification φ, as defined in Section 1, then boils down to checking the satisfiability of φ at each time step: for every time step i less than mission length m, we check if the current run of the system, computation π, satisfies formula φ. There are currently no robust, publicly-available tools for MTL satisfiability checking. For the related question of LTL satisfiability checking, we have seen that creating a tool that correctly evaluates $\pi, 0 \models \varphi$, e.g., returns a singular evaluation of satisfied or not satisfied from the first time step in the input trace, is very difficult [22–24]. For example, all eleven tools benchmarked for LTL satisfiability via explicit model checking returned at least one wrong answer and some performed worse than random guessing, as shown in Figure 2. We also found that many tools crashed or died gracelessly when an input could not be handled, or failed to distinguish between unsatisfiable verdicts and error cases. Therefore, it is important to learn from this history and emphasize correct output in RV.

Here is a scheme for scoring tools based on correctness that, differently from [4], addresses cyber-physical systems (e.g., does not explicitly consider software crashes), evaluates correctness over execution sequences, and emphasizes the criticality of incorrect verdicts. For an execution sequence produced by a given RV tool $\langle T_\varphi \rangle = \langle T_\varphi^0 \rangle, \langle T_\varphi^1 \rangle, \langle T_\varphi^2 \rangle, \langle T_\varphi^3 \rangle, \ldots$, we define the correctness score C for a benchmark of mission time $t_m \in \mathbb{N}_0$ as $C = \Sigma_{i=0}^{m} C_i$ where:

- $C_i = 10$ for a correct verdict such that $\langle T_\varphi^i \rangle.v$ is **true** if T_φ^i holds and $\langle T_\varphi^i \rangle.v$ is **false** if T_φ^i does not hold.
- $C_i = 1$ for an "I don't know" verdict such as $\langle T_\varphi^i \rangle.v$ being **maybe**, because this verdict shows liveness, and does not adversely affect system operation.
- $C_i = 0$ for no answer/failure to answer (due to a tool bug, poor responsiveness, or other reason); this verdict does not cause harm but does not instill confidence that the checker

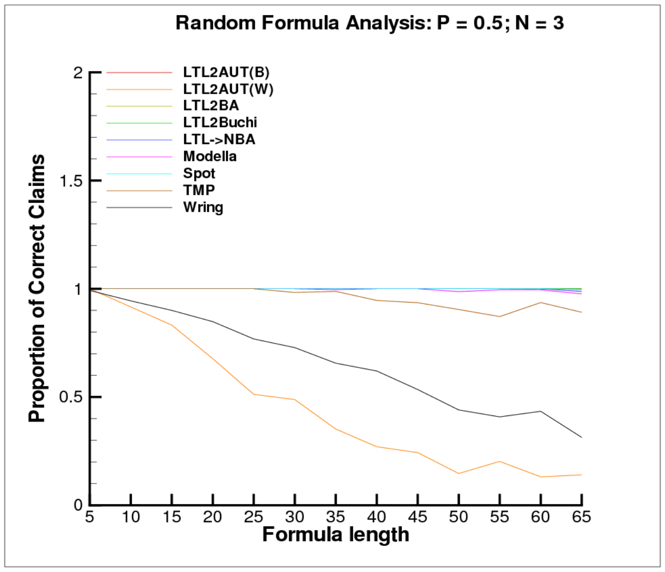

Figure 2: Graph from [21,22] showing the degradation of proportion of correct claims for random LTL formulas where the probability of choosing a temporal operator, P = 0.5 and the number of variables, N = 3. This graph shows results for random formulas of lengths 5 . . . 65. None of the tools tested in this benchmark produced all correct answers (which would result in their lines overlapping the reference black line at 1 on the y-axis) though, notably, this benchmark inspired corrections. All versions of SPOT released since this test have produced all correct answers [9,10].

is still working.

- $C_i = -\infty$; disqualification, or equivalent, such as a very large number of negative points for a wrong answer.

We propose to weight the tool's correctness scores using the secondary criteria of execution time; for each C_i we apply weight w_i such that the weighted correctness score W for a benchmark of mission time $t_m \in \mathbb{N}_0$ is $W = \Sigma_{i=0}^{m} w_i C_i$ where:.

- $w_i = 10$ where correct verdict v is reported at time $t_i \geq i$ such that t_i is the first time step where there is sufficient information to determine a correct verdict for time i. Note that for past-time formulas, $t_i = i$; for future-time formulas calculating t_i is much more complicated.
- $w_i = max\left(\left(10\left(1 - \frac{j-t_i}{t_i+100}\right)\right), 1\right)$ where correct verdict v is reported at time $j \geq t_i$.
- $w_i = 1$ for no verdict/failure to answer.

An alternative to weighting the correctness scores by execution time is to separately report correctness and verdict resolution time. Given an execution sequence produced by a given RV tool $\langle T_\varphi \rangle = \langle T_\varphi^0 \rangle, \langle T_\varphi^1 \rangle, \langle T_\varphi^2 \rangle, \langle T_\varphi^3 \rangle, \ldots$, we define the verdict resolution time score V for a benchmark of mission time $t_m \in \mathbb{N}_0$ as $V = \Sigma_{i=0}^{m}\ j - \langle T_\varphi^i \rangle.\tau_e$ where correct verdict v for $\langle T_\varphi^i \rangle$ is reported at time j. For a competition we can then normalize the verdict resolution times, setting the smallest V any tool achieves for a given benchmark to 0, thus avoiding the problem of having to calculate time t_i for all times i such that t_i is the first time step where there is sufficient information to determine a correct verdict for time i.

In addition to correctness and execution time, we may evaluate overhead or resource usage. While [4] provides a compatible memory-utilization score to the evaluation criteria presented here, differently from [4], we do not consider overhead in terms of execution time, but instead in terms of other resources used, which may depend on the execution platform (software versus hardware, runtime environment characteristics). Overhead includes points for unobtrusiveness in execution, such as minimizing system modifications to retrieve runtime input streams, and the ability to work within tight resource bounds, such as FPGA gates or power. We add bonus points to correctness and timing scores for lower overhead/smaller resource footprints. Alternatively, overall scores could be sorted by classes of overhead and resource usage, so the scores would be a set of lists, one list for each overhead/resource class ranked by correctness and execution time.

3.1 On Execution Time

Hardware or CPS runtime verification can be performed either online (running during system execution and performing analysis in real time) or offline (running post-execution); previous Competitions on Runtime Verification have considered both of these categories for software [4]. For purposes of evaluation and comparison of RV tools that analyze hardware or cyber-physical systems, the distinction between online and offline monitoring can be made by how execution time is weighted in benchmarking. Since it may be difficult to impossible to comparatively evaluate runtime verification tools on their target platforms (such as aircraft or spacecraft), online verification tool benchmarking equates to simulating a system run using a pre-recorded (offline) benchmark whilst considering execution time. Figure 3 proposes a scheme for conducting an online RV competition track for cyber-physical systems.

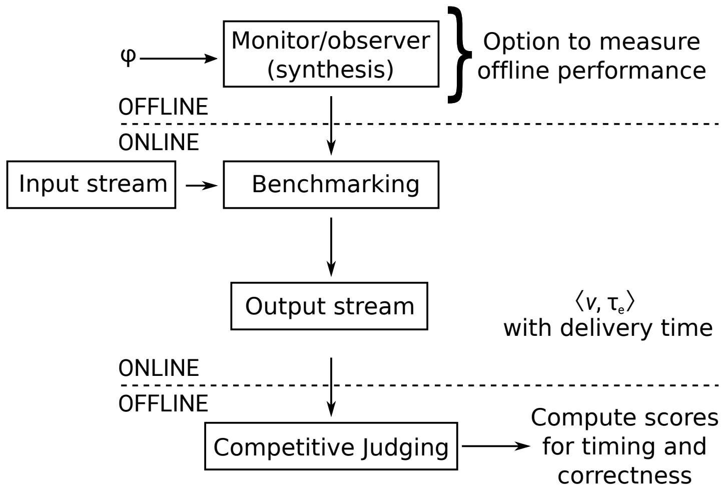
φ
Monitor/observer (synthesis)
Option to measure offline performance
OFFLINE
ONLINE
Input stream
Benchmarking
Output stream
⟨v, τe⟩ with delivery time
ONLINE
OFFLINE
Competitive Judging
Compute scores for timing and correctness

Figure 3: Online Runtime Verification for Cyber-Physical Systems Track: (1) The **requirement** φ is provided to the RV tool under test for synthesis of the monitor/observer offline, with an option to include performance characteristics of this synthesis stage in the competition score. (2) The **input stream** is streamed into the running monitor/observer in an online runtime verification phase, producing in real time the **output stream**. (3) The competition is scored utilizing the output stream (stream of verdict/time stamp tuples (v, τ_e) along with the time each tuple was reported) and any other measured performance characteristics according to the evaluation criteria (per Section 3).

When defining execution time for benchmarking RV tools, we can additionally consider the following questions:

- How do we time input streams to RV tools? Do we need to run the competition in a simulator that releases new values from pre-recorded benchmarks at specific time intervals?
- Do we consider asynchronous (event-triggered) verdicts, synchronous (time-triggered) verdicts, or a combination of both? For example, a synchronous verdict would give the most accurate answer possible at that time, but an asynchronous verdict would be expected to be maximally accurate, though necessarily delayed.
- If one purpose of RV, at least for some systems, is to provide *predictive* information, e.g, information that could be used to prevent the crash of an aircraft versus information that can be used to accurately determine the cause of a crash after the fact, then how can this be measured in competition? Necessarily, to report a probable fault before it happens, with enough time to enable fault-mitigation actions, requires a probabilistic estimation. Here we trade time for correctness: we demand tools provide a maximally-likely verdict before it is possible to evaluate whether that verdict is correct.

4 Evaluating Hardware Monitoring Tools

Realistically, there are two categories of runtime verification: software and hybrid software-hardware, aka cyber-physical. No modern system requiring RV is totally without software and there is no reason to limit RV tools that monitor hardware from taking into account software status received from the same channels, e.g., over the system bus.

It is important to note that temporal logic specifications can be used to monitor hardware as well as software. Hardware monitoring is most straightforwardly a temporal logic formula requirement evaluated over a sensor signal sent over the system bus. Hardware monitoring has the advantage that annotations, like the code instrumentation sometimes required for software monitoring, are not a possibility. Hardware monitoring also offers the challenge that specifications can grow much more expressive than for software monitoring, even incorporating partial differential equations in conjunction with temporal logic to offer a rich description of total cyber-physical system behavior.

The challenge with evaluating hardware monitoring of temporal logic specifications is chiefly one of timing: how to filter benchmark sensor data input streams and feed them through Boolean testers to populate the variables evaluated in the temporal logic specifications. Input streams with different frequencies will need to be filtered and tested in a way that is comparable. The correct verdict will depend on the correspondence between temporal logic time steps and the time divisions of input streams. One solution, which makes the competition more standardized but less realistic, would be to set up the benchmarks so that every input stream supplies values at one second intervals and one time step in temporal logic equates to one second.

Eventually, we may pair temporal logic benchmarks with partial differential equations for the evaluation of more expressive cyber-physical system monitoring tools. However, evaluating the correctness of verdicts produced for these specifications is an open area of research currently being pursued in the robotics community.

Differently from software, some platform-based evaluation criteria may serve as informative classifiers for hardware monitoring tools, with the side-effect of making them more amenable to technology transfer. Here are a few ideas for expansions of the evaluation of hardware monitoring tools.

- List tools by their platform constraints. Eventually, we may have different categories for tools with different constraints, e.g., categories for below a certain level of overhead, power, memory, bandwidth, or other constraint common to embedded platforms with limited on-board resources. Such a list would help classify tools with respect to their suitability for certain common platforms, such as UAS or CubeSats.

- Evaluate hardware tools on different platforms. Define a test set of embedded systems and run the competition separately on each of them. Candidate platforms include: FGPA, Raspberry Pi, COTS flight computer from a toy UAS.
- Test the tools with different input frequencies. Here, we foresee the benchmark being that the temporal logic requirement φ is evaluated over variables populated by Boolean testers over the input sensor data streams. For example, the altitudes from an altimeter may be an input sensor stream, filtered through a standard noise-reducing filter, and fed through the Boolean tester $(\mathit{altitude} > 600ft)$, which sets the value of a variable in φ. Varying the input stream frequencies is particularly challenging because different frequencies could result in different valuations of φ over the same dataset.

5 What Can Be Borrowed From Other Communities?

The many competitions for different aspects of design-time verification provide inspiration for the evaluation and comparison of runtime verification tools going forward.

- **Correctness must be the primary criteria for tool evaluation.** This is even more important for RV than for design-time formal techniques like model checking. During runtime, a verdict that something is wrong (or right) must be acted on immediately; there is not time to handle spurious counterexamples that happen during design time. Tools in RV, like in medicine, must be ruled by *primum non nocere* (first, do no harm); while the absence of a verdict may allow a fault to occur, the presence of an incorrect verdict can fail an otherwise properly operating system. Other competitions consider correctness; the question of what constitutes a correct answer is more complex, more nuanced, and more impactful for RV.
- **The RV competition should participate in the Federated Logic Conferences (FLoC) Olympic Games.**[7] The FLoC Olympic Games were started in 2014 in the hope of creating a tradition in the spirit of the ancient Olympic Games. Every four years, as part of the Federated Logic Conference, the Games will gather together all the challenging disciplines from a variety of computational logics in the form of the solver competitions.
- **Software benchmarks for RV could be adapted from SV-COMP.** The Competition on Software Verification[8] provides an annual snapshot of the state of the art in software verification, along with a well-organized selection of software benchmarks that are available online. A detailed competition report is published each year [5].
- **Organization is key! We need a publicly writable central repository of benchmarks.** We can also benefit from lessons learned: other communities and competitions have struggled to keep up an open, easily accessible, annually-updated repository of benchmarks that reflect the latest case studies and the current state of the art in terms of pushing performance boundaries. Examples for benchmark organization can be drawn from StarExec[9], which organizes benchmarks for several logic solver competitions, and the PRISM Bibliography[10], which collects all publicly-available artifacts from previous case studies using the PRISM model checker. The RV competition needs to design from

[7] `http://vsl2014.at/olympics/`
[8] `http://sv-comp.sosy-lab.org/2016/`
[9] `https://www.starexec.org`
[10] `http://www.prismmodelchecker.org/bib-ext.php#casestudies`

the start for a central cadre of benchmarks where authors can easily add their latest case studies and most challenging problems throughout the year, as they arise or as new papers are published.

- **Minimize specification languages; maximize freely-available translators.** This idea is mainly borrowed from HWMCC (which is standardized around AIGER [6]) and SMT-COMP (with SMT-LIB [3]), though all competitions focus on minimizing the category splitting caused by accepting too many specification languages. The availability of freely-available parsers and translators is a secondary benefit to the research community.
- **Presentation of competition results.** The established formal verification competitions have produced well-organized, extensive reports clearly organizing results in many different formats, including many different perspectives on the outcomes. The RV competition would do well to survey the best-practices from SV-COMP, HWMCC, SAT-COMP, SMT-COMP, and CASC for producing informative websites, slides, and annual reports.

6 Conclusions and Outlook

The necessity of RV will continue to grow sharply in the future; an RV competition would both help to shape the growth of tools and provide sharply needed standardization for integrating RV tools into industrial systems. The first step toward a competition should include standardizing the set of specification languages, and, at the same time, providing from the start a well-organized, publicly-writable, central repository for benchmark specifications in these formats. Thinking through the specification languages, and the organization of specifications in those languages early could have a major influence on the shape of RV on industrial platforms going forward. If the specifications are written in a language that is so simple that it is hard to make the argument to industry that the tools can reason about real-life requirements then RV tools could be hampered in technology transfer out of academia. If the benchmarks and competition are not thoughtfully constructed to emphasize correctness above all else, they will be considered too risky to try on realistic applications. Framing the competition secondarily around accurate assessments of performance, such as timing and resource usage, would provide a useful reference for transitioning the tools into practice as the best-performing tool that abides by the target systems' constraints can then be easily identified.

We are currently working on developing a graph-database of real runtime specifications from systems we have verified in the Laboratory for Temporal Logic.[11] We are constructing this database using Neo4j,[12] a publicly available, performable, NoSQL graph database implemented in Java and Scala that efficiently implements the property graph model to allow, e.g., constant-time traversals for relationships in the graph [19]. A property graph database stores Nodes (graph data records), and Relationships (directional connect nodes), with Properties (named data values of type string, number, Boolean, or array), on both Nodes and Relationships. Our current research involves proposing a standardized graph-database schema that intuitively stores specifications expressing Boolean, temporal, and probabilistic requirements. We plan to release the database as an open-source resource for those conducting runtime verification in the future. A similar scheme, with the additional information needed to run a competition, such as the inputs and the correct verdicts of different specifications on those inputs, could serve as a foundation for organizing RV benchmarks.

Information on our recent work can be found at: `http://laboratory.temporallogic.org`.

[11] http://laboratory.temporallogic.org
[12] `https://neo4j.com`

References

[1] R. Alur and T. A. Henzinger. Real-time Logics: Complexity and Expressiveness. In *LICS*, pages 390–401. IEEE, 1990.

[2] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE TSE*, 24:156–166, 1996.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Online: `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-draft-2.pdf`, June 2015.

[4] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, et al. First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, 2017.

[5] Dirk Beyer. Software verification and verifiable witnesses. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015.

[6] Armin Biere. The AIGER And-Inverter Graph (AIG) Format Version 20071012. Technical Report FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, October 2007.

[7] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal design and safety analysis of air6110 wheel brake system. In *CAV*, 2015.

[8] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2013.

[9] Alexandre Duret-Lutz. LTL translation improvements in SPOT 1.0. *International Journal of Critical Computer-Based Systems 5*, 5(1-2):31–54, 2014.

[10] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. SPOT 2.0A Framework for LTL and\omega-Automata Manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, pages 122–129. Springer, 2016.

[11] E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT Press, 1990.

[12] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Y. Rozier. Model checking at scale: Automated air traffic control design space exploration. In *Proceedings of 28th International Conference on Computer Aided Verification (CAV 2016)*, volume 9780 of *LNCS*, pages 3–22, Toronto, ON, Canada, July 2016. Springer.

[13] Johannes Geist, Kristin Yvonne Rozier, and Johann Schumann. Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In *Proceedings of the 14th International Conference on Runtime Verification (RV14)*, volume 8734, pages 215–230. Springer-Verlag, September 2014.

[14] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[15] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, volume 3253, pages 152–166. Springer, 2004.

[16] Cristian Mattarei, Alessandro Cimatti, Marco Gario, Stefano Tonetta, and Kristin Y. Rozier. Comparing different functional allocations in automated air traffic control design. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2015)*, Austin, Texas, U.S.A, September 2015. IEEE/ACM.

[17] C. Muñoz, V. Carreño, and G. Dowek. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Engineering of Fault-Tolerant Systems*, volume 4157 of *LNCS*, pages 306–325, 2006.

[18] Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science (LNCS)*, pages 357–372. Springer-Verlag, April 2014.

[19] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In *Proceedings of 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016)*, volume 9971 of *LNCS*, pages 1–19, Toronto, ON, Canada, July 2016. Springer-Verlag.

[20] K.Y. Rozier. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review Journal*, 5(2):163–203, May 2011.

[21] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[22] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):123 – 137, March 2010.

[23] K.Y. Rozier and M.Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *17th International Symposium on Formal Methods (FM2011)*, volume 6664 of *Lecture Notes in Computer Science (LNCS)*, pages 417–431. Springer-Verlag, 2011.

[24] K.Y. Rozier and M.Y. Vardi. Deterministic compilation of temporal safety properties in explicit state model checking. In *8th Haifa Verification Conference (HVC2012)*, volume 7857 of *Lecture Notes in Computer Science (LNCS)*, pages 243–259. Springer-Verlag, 2012.

[25] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *COMPASS*, pages 77–88. IEEE, 1996.

[26] Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming Journal*, 96(3):337–353, December 2014.