FROM SIMULATION TO RUNTIME VERIFICATION AND BACK: CONNECTING SINGLE-RUN VERIFICATION TECHNIQUES

Kristin Yvonne Rozier

Department of Aerospace Engineering, Iowa State University
537 Bissell Road, Ames, IA, USA
kyrozier@iastate.edu

ABSTRACT

Modern safety-critical systems, such as aircraft and spacecraft, crucially depend on rigorous verification, from design time to runtime. Simulation is a highly-developed, time-honored design-time verification technique, whereas runtime verification is a much younger outgrowth from modern complex systems that both enable embedding analysis on-board and require mission-time verification, e.g., for flight certification. While the attributes of simulation are well-defined, the vocabulary of runtime verification is still being formed; both are active research areas needed to ensure safety and security.

This invited paper explores the connections and differences between simulation and runtime verification and poses open research questions regarding how each might be used to advance past bottlenecks in the other. We unify their vocabulary, list their commonalities and contrasts, and examine how their artifacts may be connected to push the state of the art of what we can (safely) fly.

Keywords: runtime verification, monitoring, formal methods, online, stream-based.

1 INTRODUCTION

As we develop increasingly capable and complex safety-critical systems, like aircraft and spacecraft, we fuel the verification arms race: achieving flight certification presents a continuous challenge for verification techniques to keep up. Simulation is a well-established, time-honored design-time verification technique that dates back to the 1940's, during WWII, with conferences, specialized modeling languages, and a textbook established by the 1960's (Shinde 2000). In contrast, runtime verification (RV) (Havelund and Rosu 2019, Leucker and Schallhart 2009, Falcone, Havelund, and Reger 2013, Bartocci, Falcone, Francalanza, and Reger 2018) first broadly established its name at a 2001 workshop, which grew to an international conference in 2010 (Havelund and Rosu 2019), half a century later than simulation! RV compliments design-time verification techniques like simulation by carrying the verification effort later in the system development cycle to deployment time: by checking that real executions of the deployed system satisfy its requirements, we can react to off-nominal events that may not be anticipated during design-time verification. RV faces different challenges than simulation, such as the constraints of executing in real time on-board an embedded system without violating flight certification. As a field, RV is still forming a unified vocabulary; RV tools are not yet comparable with each other much less other tools and techniques for verification. Yet, despite its youth, RV has already demonstrated great promise, contributing critical verification capabilities on-board (Pike, Goodloe, Morisset, and Niller 2010, Reinbacher, Rozier, and Schumann 2014, Geist, Rozier, and Schumann 2014, Moosbrugger, Rozier, and Schumann 2017). Both techniques base their analysis on single runs of the system under verification but their relationship to each other remains unclear.

SpringSim-ANSS, 2019 April 29-May 2, Tucson, AZ, USA;

Runtime verification is a lightweight formal method that rigorously analyzes system executions in a provably-correct way, like the formal methods of model checking and theorem proving, but with the caveat that RV reasons about only one execution, rather than the exhaustive set of all possible executions examined by formal methods. RV takes as input a temporal specification depicting a fault signature, and a system execution; its output is a verdict indicating whether the system execution exhibits the fault signature. Both *trace-driven simulation* and runtime verification analyze single system runs, or executions, producing verification verdicts via the information contained in a model or specification of the relevant aspects of the system under verification. Simulation directly executes the input model, producing a large set of executions over which we compute output statistics. For the purposes of this paper, we consider trace-driven, *discrete-event stochastic simulation* because it is the most comparable to the techniques for runtime verification used on-board aerospace systems, though, we note connections between RV and other categories of simulation.

The contributions of this paper include a comparison between Simulation and RV, unifying their vocabulary, and enumerating specific points of similarity and differences that create opportunities for cross-pollination of research advances between the two. We then pose open research questions where they can feed each other, bridging the gap between verification artifacts from simulation that can be used to mature the field of RV, and artifacts from RV that can robustify simulation.

2 SIMULATION VS RUNTIME VERIFICATION

2.1 Simulation

We focus our connection between single-run techniques for verification on *discrete-event stochastic simulation*; we call this *simulation* throughout the rest of this paper. This system-design-time technique includes the modeling, simulation, and analysis of the performance of discrete-event stochastic systems; see Figure 1 for a summary.

Definition 1. *(Leemis and Park 2006) A* **discrete-event simulation model** *is defined by three attributes:*

- **stochastic** – *at least some of the system state variables are random;*
- **dynamic** – *the time evolution of the system state variables is important;*
- **discrete-event** – *significant changes in system state variables are associated with events that occur at discrete time instances only.*

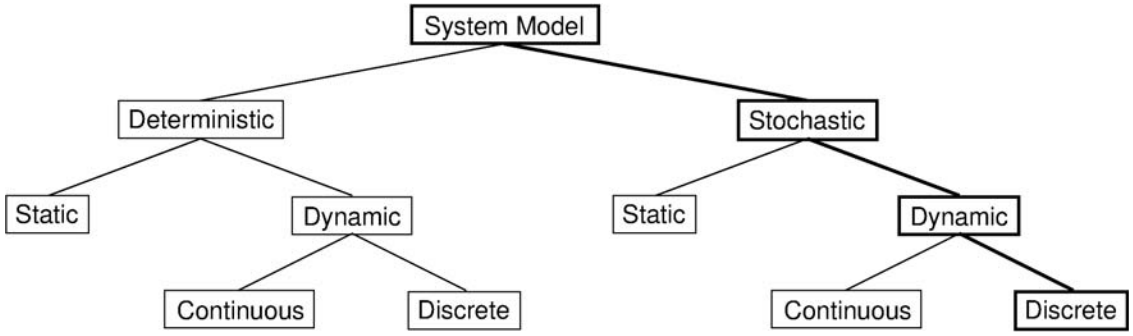

Figure 1: Characterization of a simulation system model in the form of a tree diagram from (Leemis and Park 2006). We focus on the right-most branch of the tree, as summarized by Definition 1.

On a high level, both discrete-event stochastic simulation and RV follow the same modeling algorithm (which appears as Algorithm 1.1.1 for *developing a discrete-event simulation model* in (Leemis and Park 2006)).

Definition 2. Algorithm *(Leemis and Park 2006):* **Developing a System Model**. *Perform Step 1, then iterate through Steps 2 through 6 until a valid computational model (executable hardware or software) has been created. The resulting discrete-event simulation model should be as simple as possible, but never simpler.*

1. *Decide on the goals and objectives of the system to be analyzed, usually in the form of Boolean analysis, such as deciding which faults need to be detected and which fault signatures to consider.*
2. *Based on (1), build a* **conceptual model** *of the system, defining the state variables and their relationships.*
3. *Turn the conceptual model into a* **specification model***, e.g., through collecting and analyzing data or otherwise deriving a representative model of the relevant behaviors of the system under test.*
4. *Turn the specification model into a* **computational model***, in the form of executable hardware or software.*
5. *Verify that the computational model is correct.*
6. *Validate that the computational model is consistent with the system under test.*

Related to Definition 2 Step 3's conceptual model, the following two definitions hold for both simulation and runtime verification; versions of these appear as Definitions 5.1.1 and 5.1.2 in (Leemis and Park 2006) as well as nearly every paper on runtime verification.

Definition 3. *A* **system state** *is a complete characterization of the system at an instance in time, usually represented as an assignment to the complete set of system variables.*

Definition 4. *An* **event** *is an occurrence that changes the system state, e.g., by altering the assignment to the system variables. Only an event can result in a state change.*

We consider events that do not change the system state to be excluded from the system model abstraction on purpose: they are not relevant to the characteristics of the system we wish to analyze.

Definition 5. *A* **fault** *is a deviation between the behavior in a system execution and the expected behavior, as defined by safety requirements; it can occur in either software, hardware, or a combination of the two as system states can be composed of variables representing both hardware and software. A fault can take the form of a system state that should be unreachable, or a path through multiple system states that should not be followed.*

As (Leucker and Schallhart 2009) explains, a fault might lead to a failure, but not necessarily. Differently, an error is a mistake made by a human that results in a fault and possibly in a failure.

2.2 Runtime Verification (RV)

We focus our connection between single-run techniques for verification on *online, stream-based discrete-time runtime verification*; see Figure 2 for a summary comparable to Figure 1. Similarly, we refer to this branch of RV throughout the paper. Our choice is motivated by two major factors: this is the branch of RV with the richest connections to simulation; these branches of simulation and RV are the most practical as both are commonly-used for avionics certification such as constructing safety cases required by the FAA's DO-178-B (RTCA 1992) DO-178-C (RTCA 2012), and DO-254 (RTCA 2000).

Definition 6. Online, stream-based discrete-time runtime verification *is defined by three attributes:*

- **online** – *the RV engine executes at the same time (synchronously or asynchronously) with the system under verification, during deployment, observing the current execution and part of its history*

- **stream-based** – *the output of the RV engine is a stream of* $\langle time, verdict \rangle$ *tuples, evaluating for every time t in the finite system execution whether the execution starting at time t satisfies the monitored requirement (as opposed to outputting a single such verdict for the total execution from start to finish);*
- **discrete-time** – *fault signatures are described by temporal logic formulas (or, equivalently, automata) over discrete time instances only, such as sensor signals.*

Other variations of RV include offline execution, when the RV engine analyzes a complete system execution recorded in a log file. Offline RV avoids the constraints that present challenges for online RV, including real-time execution, operational constraints of running on-board the system under verification, and unobtrusiveness constraints for non-interference with certified sub-systems or resources such as power, cost, CPU cycles, or overhead.

Comparably to discrete-event stochastic simulation (Definition 1), runtime verification reasons about systems that are:

- **stochastic** – at the very least the environment is always a stochastic component of the system that we have to address, and the system faults RV aims to detect are largely random failures;
- **dynamic** – RV reasons over temporal traces of system execution;
- **discrete-event** – while there are some RV specification logics that reason over continuous time (e.g., Signal Temporal Logic (Maler and Nickovic 2004)), the sensor signals that serve as input to RV engines are inherently discrete.

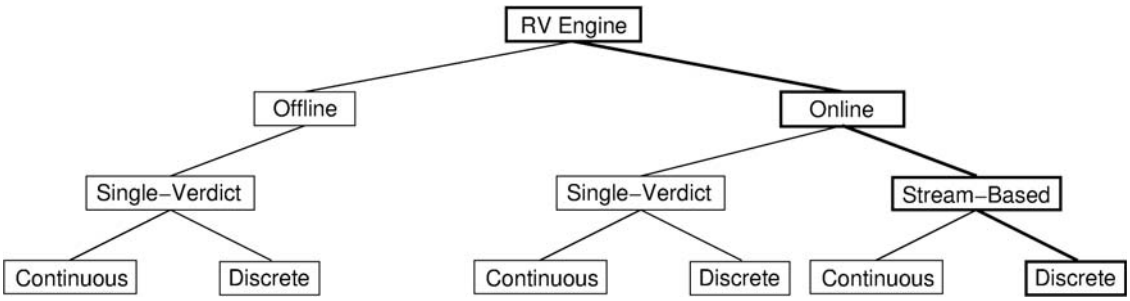

Figure 2: Characterization of a runtime verification engine paradigm in the form of a high-level tree diagram comparable to 1. We focus on the right-most branch of the tree, as summarized by Definition 6.

Definition 7. Algorithm *(Bartocci, Falcone, Francalanza, and Reger 2018):* **Developing a Runtime Verification Engine**. *Perform three steps, iterating until the required robustness (verification and validation) targets for the system under verification have been achieved.*

1. **Specifying (Un)Desired System Behavior** – *akin to Algorithm 2 Steps 1-3, including defining the verification objectives, identifying system variables and the correct level of abstraction, and faithfully representing those in a specification model.*
2. **Producing a Monitor from a Specification** – *akin to Algorithm 2 Steps 4-6, including producing from the specification model an executable runtime verification engine, implemented in either software or programmable hardware (e.g., FPGAs), formally proving its correctness, and validating this engine.*
3. **Connecting a Monitor to a System** – *execution of an RV engine running in parallel with the system depends upon connecting input signals, e.g., from on-board sensors or software, filtering, creating Boolean testers, or otherwise pre-processing these signals, and ensuring execution of the RV engine obeys power, weight, timing, or other unobtrusiveness constraints. RV engines may also be able to trigger system actions, such as executing a mitigation plan in the case of a detected fault.*

2.3 Commonalities

Common Objectives. (See Algorithm 2, Step 1.) Certainly simulation can serve many objectives, including many definitions of performance and safety. Simulation overlaps with the primary purpose of runtime verification in the areas of *correctness* (always behaving the way we expect), *safety* (not violating requirements), and *performance-as-safety* (upholding minimum requirements for safe real-time operation such as responsiveness or throughput).

Computational Model Format. Most importantly, discrete-event simulation and runtime verification share significant overlap in the space of conceptual model formats (corresponding to Algorithm 2 Step 4). Both reason about single executions of the system, also called *traces* or *runs*, though they differ in the analysis performed on these. Both are most commonly aimed for software implementations (Delgado, Gates, and Roach 2004), though runtime verification may also require implementation in programmable hardware for deployment on-board real systems (Meredith, Jin, Griffith, Chen, and Roşu 2012, Reinbacher, Rozier, and Schumann 2014, Rozier and Schumann 2017).

Notion of Time. Related to the conceptual model, there is significant overlap in the way discrete-event simulation and runtime verification can represent time and the transition between system states. Both discrete-event simulation and runtime verification can be either event-triggered or time-triggered, with event-triggered being the most common, as in next-event simulation. Few RV tools are time-triggered by default (Falcone, Krstić, Reger, and Traytel 2018), with a few notable exception of time-triggered monitors (Rozier and Schumann 2017, Azzopardi, Colombo, Ebejer, Mallia, and Pace 2017, Navabpour, Joshi, Wu, Berkovich, Medhat, Bonakdarpour, and Fischmeister 2013). Importantly, runtime verification tools do not generally represent clocks in the same way as simulations, though we can usually define equivalences, e.g., through counters or reliance on checks of external-to-the-RV-engine system clocks.

Verification. (See Algorithm 2, Step 5.) Verification of a computational model for simulation often involves extensive testing, with respect to applicable coverage metrics and a comparison to expected outcomes defined by the human modeling the system. While this is one often-used form of verification for RV engines, the formal nature of RV allows for more rigorous, and therefore more automated, verification efforts as well. For example, it is possible to create a test-case for RV with an a priori known exact correct answer.

Validation. (See Algorithm 2, Step 6.) In the case that the system under test is not yet operational, validation for simulation and RV is based mostly on consistency checks: do changes in the specification result in the expected changes in the outputs? In the case that the system under test is operational, both techniques utilize comparison with individual system runs on the real system for validation.

2.4 Contrasts

Purpose. The purpose of simulation is insight (Leemis and Park 2006) whereas the purpose of RV is fault detection (Leucker and Schallhart 2009). Simulation is a design-time verification technique, executed offline, and used to improve the system before deployment. Insights include characterizing system performance, understanding how a subtle system feature works, discovering component interactions, improving actions in response to system operations, or maximizing profit during system deployment. RV is a runtime verification technique used to detect deviations from nominal operation online, in real time, to enable mitigation actions. Its purpose is early-as-possible identification or prediction of faults or partial fault signatures in order to robustify system operation, ensure safety/compliance with requirements, and enable certification, e.g., flight certification from the FAA. Consequently, their *usage* differs: simulation analysis algorithms analyze a set of runs to find trends, whereas RV analyzes only the one, current run to identify faults as early as possible, preferably *before* executing the whole run.

Modeling/Specification Language. (See Algorithm 2, Step 3.) A simulation refers to a computational model, nearly always coded in a programming language; the system is modeled explicitly in that the computational model emulates its behavior. A simulation model is in either a general purpose language (most typically C/C++), or a specialized simulation language (Law, Kelton, and Kelton 1991). RV translates a temporal logic specification into an executable monitor, executing within some RV engine. RV has its own extensive proliferation of specialized specification languages (Falcone, Krstić, Reger, and Traytel 2018). In RV, the system model is implied by the consequences of the specification: an incorrect, off-nominal, or unexpected execution will violate the specification. Both simulation and RV model as little as necessary to capture the relevant details of the system and depend critically on abstraction for this. Some RV specifications more directly pull abstractions from a simulation model, e.g., LOLA (Faymonville, Finkbeiner, Schirmer, and Torfah 2016), whereas others are pure logic formulas. Simulation runs are often pictured as timelines and the *conceptual model* often includes timelines of different scenarios in the model. This suggests a promising relationship because for every RV specification in the most-common family of linear temporal logics, there exists a timeline equivalent to that specification that may be derived directly from a specification model. There is potential for turning the modeling/specification language contrast into a commonality via focusing on dual-use specifications, and unifying the expressive common-cores of languages for simulation and RV to enable modeling a system once but utilizing both verification techniques.

Outputs. The output statistics of simulation characterize system executions in the aggregate. They include job- (or customer- or event-) averaged statistics like average delays, average interarrival times, or average service times, as well as time-averaged statistics such as utilization. RV instead outputs something much simpler; a typical monitor checks that one system execution (the current one) satisfies a (temporal) safety requirement and outputs the corresponding Boolean verdict. Opportunities for validation of simulation via RV include using RV for extracting statistics, e.g. http://www.cs.um.edu.mt/svrg/Tools/larvastat/, and new extensions to RV that reason about sets of traces, encapsulated as hyperproperties (Finkbeiner, Hahn, Stenger, and Tentrup 2017, Bonakdarpour, Sánchez, and Schneider 2018). Certainly simulations could generate traces with Boolean-valued properties to validate RV frameworks, though one challenge that presents is how to efficiently generate traces with known RV verdicts, aka oracles.

Deployment. Simulation is chiefly a design-time verification technique, most often performed on a system model before the final system is built, whereas RV is a mission-time verification technique, most often deployed on the fully-operational system. Opportunities for bridging these two verification techniques include utilizing runtime verification during design time, or using the results of RV from the last system release to inform the model of the next one.

3 RESEARCH QUESTIONS

3.1 From Simulation to Runtime Verification

Simulation results in output statistics that characterize system executions in the aggregate: both job-(or customer- or event-)averaged statistics like average delays, average interarrival times, or average service times, as well as time-averaged statistics such as utilization. Simulation is inherently probabilistic in nature, in contrast to the Boolean nature of RV, which takes as input a temporal logic requirement and a single system execution and outputs a verdict as to whether the execution upholds the requirement. However, we may be able to utilize the outputs of simulation to mitigate the biggest challenge facing RV.

Arguably, the biggest challenge to the deployment of RV is the input specification bottleneck (Rozier 2016). RV critically depends on the input specification being monitored, φ. If φ is too restrictive, the RV engine will "cry wolf" and output false alerts; if φ is too permissive, exactly the critical faults RV was deployed to catch will go undetected. While φ represents a characterization of a single execution with respect to some

subset of the system state variables, one reason φ is so hard to write is because it is based on an implied model of the system, usually in the author's head.

Both simulation and RV are defined over system states and state variables; call this set Σ. Per Algorithm 2, we create the simulation specification model M from Σ and, via iterative cycles of implementation, verification, and validation, derive from M a simple-as-possible computational model that accurately simulates the pertinent system behaviors. Executing the computational model produces a single system trace; a simulation study generates a large set of single-run executions, varying assignments to Σ over time, until we have have enough data to calculate trustworthy output statistics. Importantly, to the extent that we can interpret simulation output statistics as pattern-like characterizations of a single execution trace, we can use them to inform φ, the critical input to any RV engine, as Figure 3 illustrates.

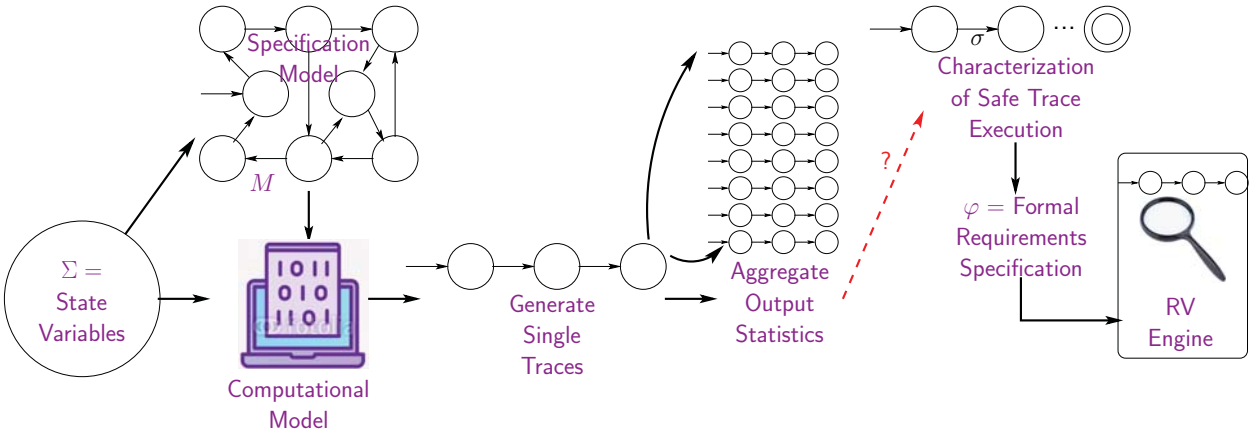

Figure 3: Possible workflow for connecting the outputs of simulation to the inputs for runtime verification: if we can formalize and automate the translation of simulation output statistics to supply the requirements from which we create runtime monitors, we can mitigate the biggest bottleneck in RV.

A scant-explored but promising research direction would be defining an automated translation from various output statistics to φ. For example, a simulation of an automated air traffic control system serves to demonstrate with appropriately high confidence that communication delays never exceed a critical threshold, and may also help define that threshold by characterizing system behavior with varying threshold values. Intuitively, we can translate this information into a temporal logic specification φ, for example monitoring that an aircraft with a predicted loss of separation always receives a resolution communication from air traffic control within the time threshold. The RV engine could then send an alert if not, triggering a mitigation action such as a maneuver by the other aircraft.

3.2 From Runtime Verification to Simulation

Due to the known difficulties in validating a complex simulation model, simulations are chiefly validated via the execution of as many consistency checks as possible (Leemis and Park 2006), e.g., pg 1.2.7. For example, if we have a simple single-server queue, the wait time, w, of a job in the service node (queue and service) should equal the sum of the delay, d, and the service time, s. Relatedly, their averages should have the same relationship: $\overline{w} = \overline{d} + \overline{s}$. Conveniently, the specialty of RV is consistency-checking. RV could be deployed during simulation, either in parallel or instrumented into the system model, to help with validation and verification of the simulation. Runtime verification languages vary widely in their expressivity, and there is currently not a single RV engine that could perform all of the consistency checks needed to validate

and verify a simulation model, but perhaps a desire for more robust, automated, and extensive validation and verification of simulations will provide the motivation to unify these capabilities into one engine.

4 OBSERVATIONS FOR FUTURE CONNECTIONS BETWEEN SIMULATION AND RUNTIME VERIFICATION

General vs Specialized Specification Languages. Both simulation and RV have on-going, at times religious, debates on the merits of more general-purpose/generic modeling/specification languages versus specialized languages. Discrete-event simulation models are effectively implemented in either a) any general-purpose programming language suitable for scientific computing (Bratley, Fox, and Schrage 2011) or b) any number of specialized simulation languages (Law, Kelton, and Kelton 1991). RV specifications are effectively implemented in either a) a canonical specification logic enabling benchmarking and comparison of results across multiple RV tools or algorithms or b) any number of specialized logics, each currently analyzed by one RV tool/algorithm but enabling intuitive and efficient encoding of fault signatures particular to the system under test.

Continuous Time. Both trees (Figures 1, 2) have a "continuous" leaf down the branch we highlighted in this paper; those may offer further interesting connections. Though continuous-time runtime verification is less-often used in aerospace applications, there are several runtime verification engines that can take into consideration the physical (in addition to logical) system time: (Basin, Krstic, and Traytel 2017, Rapin 2017, Drabek and Weiss 2017, Colombo and Pace 2017, Reger, Cruz, and Rydeheard 2015, Basin, Klaedtke, and Zalinescu 2017, Luo, Zhang, Lee, Jin, Meredith, Şerbănuţă, and Roşu 2014, Dou, Bianculli, and Briand 2017, Azzopardi, Colombo, Ebejer, Mallia, and Pace 2017).

Connections to Runtime Enforcement and Predictive RV. Runtime Enforcement compares the current execution to a required safety property like RV, but in the case of failure, also produces a revised sequence of input events that satisfy the monitored requirement (Falcone 2010). The output of runtime enforcement, the minimum modifications needed to ensure the requirement holds, may be adaptable to serve as modifications to a simulation system model. Similarly, model-predictive runtime verification (MPRV) predicts the most-likely future system state based on the trace-so-far and a system model that has been verified via simulation (Zhang, Li, Zambreno, Jones, and Rozier 2019). Predictions from MPRV should match the most-likely statistics from simulation output analysis and could be used to update the simulation model, just as the simulation model necessarily carries over to the MPRV engine; it would be a promising research direction to develop this bi-directional infrastructure.

ACKNOWLEDGMENTS

Thanks to Professors Phillip H. Jones, Giles Reger, and Joseph Zambreno for constructive feedback on earlier drafts of this paper. Work supported in part by NSF CAREER Award CNS-1552934, NASA ECF NNX16AR57G, and NSF PFI:BIC grant CNS-1257011.

REFERENCES

Azzopardi, S., C. Colombo, J.-P. Ebejer, E. Mallia, and G. J. Pace. 2017. "Runtime Verification using Valour".

Bartocci, E., Y. Falcone, A. Francalanza, and G. Reger. 2018. "Introduction to runtime verification". In *Lectures on Runtime Verification*, pp. 1–33. Springer.

Basin, D., F. Klaedtke, and E. Zalinescu. 2017. "The MonPoly monitoring tool". In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, Volume 3, pp. 19–28. EasyChair.

Basin, D. A., S. Krstic, and D. Traytel. 2017. "AERIAL: Almost Event-Rate Independent Algorithms for Monitoring Metric Regular Properties.". In *RV-CuBES*, pp. 29–36.

Bonakdarpour, B., C. Sánchez, and G. Schneider. 2018. "Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification". In *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, Proceedings, Part II*, pp. 8–27.

Bratley, P., B. L. Fox, and L. E. Schrage. 2011. *A guide to simulation*. Springer Science & Business Media.

Colombo, C., and G. J. Pace. 2017. "Runtime Verification using LARVA.". In *RV-CuBES*, pp. 55–63.

Delgado, N., A. Q. Gates, and S. Roach. 2004. "A taxonomy and catalog of runtime software-fault monitoring tools". *IEEE Transactions on software Engineering* vol. 30 (12), pp. 859–872.

Dou, W., D. Bianculli, and L. Briand. 2017. "TemPsy-Check: a Tool for Model-driven Trace Checking of Pattern-based Temporal Properties". In *Proceedings of RV-CUBES 2017: an International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, pp. 64–70. EasyChair.

Drabek, C., and G. Weiss. 2017. "DANA-Description and Analysis of Networked Applications.". In *RV-CuBES*, pp. 71–80.

Falcone, Y. 2010. "You should better enforce than verify". In *International Conference on Runtime Verification*, pp. 89–105. Springer.

Falcone, Y., K. Havelund, and G. Reger. 2013. "A Tutorial on Runtime Verification.". *Engineering dependable software systems* vol. 34, pp. 141–175.

Falcone, Y., S. Krstić, G. Reger, and D. Traytel. 2018. "A taxonomy for classifying runtime verification tools". In *International Conference on Runtime Verification*, pp. 241–262. Springer.

Faymonville, P., B. Finkbeiner, S. Schirmer, and H. Torfah. 2016. "A stream-based specification language for network monitoring". In *International Conference on Runtime Verification*, pp. 152–168. Springer.

Finkbeiner, B., C. Hahn, M. Stenger, and L. Tentrup. 2017. "Monitoring Hyperproperties". In *Proceedings of the 17th International Conference on Runtime Verification*, pp. 190–207.

Geist, J., K. Y. Rozier, and J. Schumann. 2014, September. "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, pp. 215–230, Springer-Verlag.

Klaus Havelund and Gregore Rosu 2019. "Runtime Verification". Online: http://www.runtime-verification.org/.

Law, A. M., W. D. Kelton, and W. D. Kelton. 1991. *Simulation modeling and analysis*, Volume 2. McGraw-Hill New York.

Leemis, L. M., and S. K. Park. 2006. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ.

Leucker, M., and C. Schallhart. 2009. "A brief account of runtime verification". *The Journal of Logic and Algebraic Programming* vol. 78 (5), pp. 293–303.

Luo, Q., Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu. 2014. "RV-Monitor: Efficient parametric runtime verification with simultaneous properties". In *International Conference on Runtime Verification*, pp. 285–300. Springer.

Maler, O., and D. Nickovic. 2004. "Monitoring temporal properties of continuous signals". In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 152–166. Springer.

Meredith, P. O., D. Jin, D. Griffith, F. Chen, and G. Roşu. 2012. "An overview of the MOP runtime verification framework". *International Journal on Software Tools for Technology Transfer* vol. 14 (3), pp. 249–289.

Moosbrugger, P., K. Y. Rozier, and J. Schumann. 2017, April. "R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems". pp. 1–31.

Navabpour, S., Y. Joshi, W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. 2013. "RiTHM: a tool for enabling time-triggered runtime verification for C programs". In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 603–606. ACM.

Pike, L., A. Goodloe, R. Morisset, and S. Niller. 2010. "Copilot: A Hard Real-Time Runtime Monitor". In *RV*, Volume 6418 of *LNCS*, pp. 345–359, Springer.

Rapin, N. 2017. "ARTiMon Monitoring Tool. The Time Domains". *Kalpa Publications in Computing* vol. 3, pp. 106–122.

Reger, G., H. C. Cruz, and D. Rydeheard. 2015. "MarQ: monitoring at runtime with QEA". In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 596–610. Springer.

Reinbacher, T., K. Y. Rozier, and J. Schumann. 2014, April. "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)*, pp. 357–372, Springer-Verlag.

Rozier, K. Y. 2016, July. "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*, pp. 1–19. Toronto, ON, Canada, Springer-Verlag.

Rozier, K. Y., and J. Schumann. 2017, September. "R2U2: Tool Overview". In *Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES)*, Volume 3, pp. 138–156. Seattle, WA, USA, Kalpa Publications.

RTCA 1992. "DO-178B: Software Considerations in Airborne Systems and Equipment Certification".

RTCA 2000, April. "DO-254: Design Assurance Guidance for Airborne Electronic Hardware".

RTCA 2012. "DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification".

Sagar Shinde 2000, April. "Simulation & Modeling Team: Introduction to Modeling and Simulation Systems; A. Historical Perspective". http://www.uh.edu/~lcr3600/simulation/historical.html. University of Houston.

Zhang, P., J. Li, J. Zambreno, P. H. Jones, and K. Y. Rozier. 2019. "Runtime Model Predictive Verification on Embedded Platforms". *Under submission* vol. TBD.

AUTHOR BIOGRAPHY

KRISTIN YVONNE ROZIER is an Assistant Professor of Aerospace Engineering, Computer Science, Electrical and Computer Engineering, and Mathematics at Iowa State University. She earned her Ph.D. in Computer Science from Rice University and B.S. and M.S. degrees from The College of William and Mary. Dr. Rozier's research focuses on automated techniques for the formal specification, validation, and verification of safety critical systems. Her primary research interests include: design-time checking of system logic and system requirements; runtime system health management; and safety and security analysis. Her email address is kyrozier@iastate.edu.

Her advances in computation for the aerospace domain earned her many awards including: the NSF CAREER Award; the NASA Early Career Faculty Award; American Helicopter Society's Howard Hughes Award; Women in Aerospace Inaugural Initiative-Inspiration-Impact Award; two NASA Group Achievement Awards; two NASA Superior Accomplishment Awards; Lockheed Martin Space Operations Lightning Award; AIAA's Intelligent Systems Distinguished Service Award. She is an Associate Fellow of AIAA and a Senior Member of IEEE, ACM, and SWE. Dr. Rozier serves on the AIAA Intelligent Systems Technical Committee, where she chairs the Professional Development, Education, and Outreach subcommittee. She has served on the NASA Formal Methods Symposium Steering Committee since working to found that conference in 2008.