Challenges in High-Assurance Runtime Verification

Alwyn Goodloe

NASA Langley Research Center, Hampton, Virginia, USA
a.goodloe@nasa.gov

Abstract. Safety-critical systems are growing more complex and becoming increasingly autonomous. Runtime Verification (RV) has the potential to provide protections when a system cannot be assured by conventional means, but only if the RV itself can be trusted. In this paper, we proffer a number of challenges to realizing high-assurance RV and illustrate how we have addressed them in our research. We argue that high-assurance RV provides a rich target for automated verification tools in hope of fostering closer collaboration among the communities.

1 Introduction

Safety-critical systems, such as aircraft, automobiles, and medical devices are those systems whose failure could result in loss of life, significant property damage, or damage to the environment [23]. The grave consequences of failure have compelled industry and regulatory authorities to adopt conservative design approaches and exhaustive verification and validation (V&V) procedures to prevent mishaps. In addition, strict licensing requirements are often placed on human operators of many safety-critical systems. In practice, the verification and validation of avionics and other safety-critical software systems relies heavily on system predictability; and existing regulatory guidance, such as DO-178C [30], do not have provisions to assure safety-critical systems that do not exhibit predictable behavior at certification. Yet technological advances are enabling the development of increasingly autonomous (IA) cyber-physical systems (CPS) that modify their behavior in response to the external environment and learn from their experience. While unmanned aircraft systems (UAS) and self-driving cars have the potential of transforming society in many beneficial ways, they also pose new dangers to public safety. The algorithmic methods such as machine learning that enable autonomy lack the salient feature of predictability since the system's behavior depends on what it has learned. Consequently, the problem of assuring safety-critical IA CPS is both a barrier to industrial use and a significant research challenge [10].

Runtime verification (RV) [15], where monitors detect and respond to property violations at runtime, has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. Technically speaking, a RV monitor takes a logical specification ϕ and execution trace τ of state information of the system under observation (SUO) and decides whether τ satisfies ϕ. The *Simplex Architecture* [33] provides a model architectural pattern for RV,where a monitor checks that the executing SUO satisfies a specification and, if the property is violated, the RV system will switch control to a more conservative component that can be assured using conventional means that *steers* the system into a safe state. *High-assurance* RV provides an assured level of safety even when the SUO itself cannot be verified by conventional means.

Contributions: During the course of our research we have been guided by the question "what issues must be addressed in a convincing argument that high-assurance RV can safeguard a system that cannot be otherwise assured?" In this paper, we chronicle a number of challenges we have identified that must be thoroughly addressed in order to actualize high-assurance RV. We hope that this helps inform other researchers that wish to apply RV to safety-critical systems. We examine how these issues have been addressed in our work on the Copilot RV framework [26, 28]. A theme of our research has been the application of lightweight formal methods to achieve high-assurance and we identify opportunities for closer collaboration between the RV and tool communities.

2 Copilot

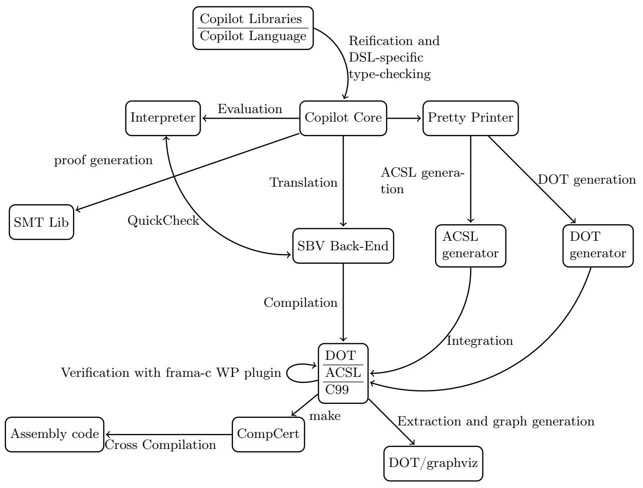

Fig. 1. The Copilot toolchain.

Copilot is an RV framework targeted at safety-critical hard real-time systems, which has served as an experimental platform enabling a research program in high-assurance RV. Copilot is a domain specific language embedded (EDSL) in the functional programming language Haskell tailored to programming monitors for hard real-time, reactive systems.[1]

Copilot is a *stream* based language where a stream is an infinite sequence of values that must conform to the same type. All transformations of data in Copilot must be propagated through streams. Copilot guarantees that specifications compile to constant-time and constant-space implementations. Copilot streams mimic both the syntax and semantics of Haskell lazy lists with the exception that operators are automatically promoted point-wise to the list level.

Two types of temporal operators are provided in Copilot, one for delaying streams and one for looking into the future of streams:

```
(++) :: [a] → Stream a → Stream a
drop :: Int → Stream a → Stream a
```

Here `xs ++ s` prepends the list `xs` at the front of the stream `s`. The expression `drop k s` skips the first `k` values of the stream `s`, returning the remainder of the stream.

Copilot's toolchain is depicted in Figure 1. A Copilot program is reified (i.e., transformed from a recursive structure into explicit graphs) and then some domain-specific type-checking is done. At this point, we have transformed the program into the "core" language, an intermediate representation. The core package contains an interpreter that can be viewed as an operational semantics for the language. The back-end translates a Copilot core program into the language of another Haskell-hosted EDSL, Symbolic Bit Vectors (SBV)[2], which we use to generate C monitors.

3 From Whence the Specification

Sound systems engineering practices as well as regulatory guidelines typically mandate safety-critical systems have detailed written requirements as well as a thorough safety assessment. Safety is a system level property, so if RV is to provide safety guarantees, then the monitor specifications must flow down from the requirements and safety analyses. Indeed, Rushby's study [31, 32] demonstrated that a convincing safety case for an IA system protected by RV demands evidence that the monitor specifications are derived from validated requirements and that the monitor specifications should include checks of the assumptions on which safe operation of the system rests are indeed satisfied.

Challenge: Monitor specifications should derive from system level requirements and assumptions that have been validated by domain experts.

As machine intelligence replaces people, RV is likely to be called upon to enforce safe behaviors that humans began learning in early childhood. For an autonomous automobile controlled by machine learning, a safety property might be

[1] `https://github.com/Copilot-Language`

[2] `http://hackage.haskell.org/package/sbv`, BSD3 license.

"do not hit a pedestrian" or "do not behave erratically", but what do these statements mean precisely? A more precise statement such as "maintain five meters distance from any object in the vehicle path" may be more formal, but is that what the expert wanted? Phrases like "erratic behavior" may seem reasonable to a mature adult, but formalizing such statements can be an open ended problem.

Challenge: Precisely formalize safety properties in a logic.

Copilot Approach: We are conducting case studies informed by collaborations with colleagues who are developing new concepts that will enable aircraft to perform autonomous flight by self-optimizing their four-dimensional trajectories while conforming to constraints such as required times of arrival generated by air-traffic service providers on the ground. Many of the proposed algorithms [21] do not behave with the predictability of conventional systems. Consequently, it is not possible to provide the required level of assurance that the newly computed trajectories preserve safe aircraft separation. The separation requirement for two aircraft is specified by a minimum horizontal separation D (typically, 5 nautical miles). Fortuitously, colleagues at NASA have discovered analytical formula, called *criteria* [25], that characterize resolution maneuvers that both ensure safe separation when one aircraft maneuvers and ensures separation when two conflicting aircraft both maneuver. The criteria have been extensively validated by domain experts who conducted sophisticated simulations as well as performing formal mathematical proofs using the Prototype Verification System (PVS) theorem prover. We have encoded these conditions as Copilot specifications. The criteria for horizontal separation for two aircraft is given as follows:

$$\texttt{horiz\_criteria}(\boldsymbol{s}, \epsilon, \boldsymbol{v}) \equiv \boldsymbol{s} \cdot \boldsymbol{v} \geq \epsilon \frac{\sqrt{\boldsymbol{s} \cdot \boldsymbol{s} - D^2}}{D} \det(\boldsymbol{s}, \boldsymbol{v}) \\ \wedge \epsilon \det(\boldsymbol{s}, \boldsymbol{v}) \leq 0$$

where $\boldsymbol{s}$ is the relative position vector for the two aircraft, ϵ is 1 or -1, and $\boldsymbol{v}$ is the relative velocity vector after a planned maneuver. The position is assumed to be given in Earth Centered Earth Fixed (ECEF) coordinates. This is easily encoded in Copilot's EDSL as:

```haskell
hor_rr :: Stream Double → Stream Double → Stream Double
hor_rr sx sy= (sqrt $ (normsq2dim sx sy) - (minHorSep ∗ minHorSep)
    ) / (minHorSep)

horizontalCriterionForConflictResolution :: Stream Double →
    Stream Double → Stream Double → Stream Double → Stream
    Double → Stream Bool
horizontalCriterionForConflictResolution sx sy e vx vy = ((
    scalar2dim sx sy vx vy) ≥ (e ∗ (hor_rr sx sy) ∗ (det2dim sx
    sy vx vy)) ) && ( ((det2dim sx sy vx vy) ∗e)  ≤ 0.0)
```

Formalizing the assumptions about the reliability of communicating aircraft position data is ongoing work. A case study formalizing what it means for a UAS to behave erratically is planned as future work.

4 Observability

Guaranteeing that all the data required by the specification is actually *observable* is one of the principal engineering challenges of RV. In embedded systems, the RV specification often involves data from a number of different types of data sources, including state data of executing programs, sensor data, as well as data that is communicated from other systems. The safety properties of cyber-physical systems are often formulated by aerospace and automobile engineers that are domain experts, but can have varying degrees of understanding of the computing systems, so the RV engineer needs to be very proactive in addressing the observability issue. In embedded systems, the closed nature of the hardware platforms and proprietary issues can make it impossible to observe the information required in the specification. Additional sensors may be needed or communication data formats changed. At times it is necessary to change the specification so that it only depends on observable data. The observability issue may seem like an "engineering detail", but based on our experience, it is often a significant obstacle resulting in delays, frustration, and sometimes preventing progress altogether.

Challenge: Determining observability of the state and environment variables in the specification.

Copilot Approach: How a RV framework obtains state data impacts the properties that can be monitored. Many RV frameworks such as MAC [22] and MOP [8] instrument the software and hardware of the SUO so that it emits events of interest to the monitor. While attractive from the viewpoint of maximizing state observability, the additional overhead may affect worst case execution times and consequently the scheduling; regulatory guidelines may also require recertification of that system. Copilot and several other RV frameworks [6, 13, 20] opt to sacrifice complete observability by sampling events. Copilot monitors run as dedicated threads and sample program variables and state information via shared memory. Currently, we rely on scheduling analysis and experimentation to ensure that we sample values at a sufficient rate that specification violations are detected. This has been very successful when the implementation platform is running a real-time operating system (RTOS) with deterministic scheduling guarantees, but we cannot make strong assertions of efficacy running on less specialized systems.

A critical lesson learned in the course of conducting many case studies is to ask questions about observability early and often. If monitoring the state of an executing program, is it possible that the monitor fails to detect a state change? It is often necessary to read sensor data to obtain the required state data (e.g.

aircraft pitch and vehicle position) or environmental data (e.g. temperature). If it is raw sensor data, do we apply filters before feeding the data into the monitors? Is the data available in the same coordinate systems demanded of the specification? Can we ensure the integrity and provenance of the data being observed?

The aircraft safe separation criteria specification introduced in Section 3 requires the monitor to observe state data for both the aircraft the monitor is executing on as well as the "intruder" aircraft. Hence, the monitors must sample data from executing programs (planned maneuver), onboard positioning sensors, and data broadcast from other vehicles.

5 Traceability

To ensure that the requirements and safety analyses performed early in systems development are reflected throughout the lifecycle, many guidelines for safety-critical software, such as DO-178C, require documentation of traceability from requirements to object code. Consequently, to promote the acceptance of high-assurance RV, the monitor generation frameworks should produce documentation that supports traceability from specification to monitor code.

Challenge: Support traceability from the requirements and system level analysis to the actual monitor code.

Copilot Approach: Using SBV to generate C monitors may create many small files and it can be quite difficult to relate this to the specification. The code generation module has recently been revised to generate documentation that improves traceability. The user can insert labels in their specifications that flow down to the documentation. The translation process creates C header files with documentation formatted to be processed by the plain text graph description language processor DOT [1]. Each C function has accompanying auto-generated graphical documentation.

In the case of the following example:

```
hor_rr sx sy  = (label "?hor_rr_dividend" $ sqrt $ (normsq2dim sx
    sy) - (minHorSep * minHorSep)) / (label "?hor_rr_divisor" $
    minHorSep)
```

the SBV translation breaks this relatively simple expression into numerous small C functions and function parameters get instantiated with the variables being monitored. The auto-generated documentation for one of these files appears similar to Figure 2, where the labels have the names of the program variables being monitored.

6 Fault-Tolerant RV

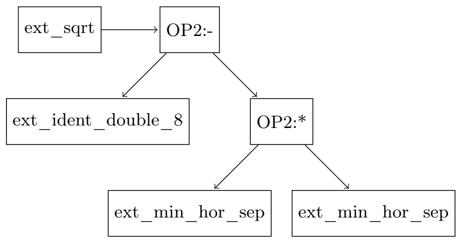

Fig. 2. Autogenerated Documentation

Safety engineers employ a range of established methods to identify hazards and failure modes [3]. The level of desired reliability determines what faults the system must be designed to tolerate. If RV is to be the guarantor of safety, then it must at least meet the level of reliability demanded of the system as a whole. Thus, high-assurance RV should be designed to be *fault-tolerant* [7], meaning it continues to provide its required functionality in the presence of faults. A fault-tolerant system must not contain a *single point of failure*.

Ideally, the RV and the SUO should not be subject to common modes of failure. For instance, software errors in the SUO such as numerical overflows and memory leaks that can render a system inoperable should not affect the RV. A fault-tolerant system must also be robust in the presence of hardware faults such as sensor failures and voltage spikes. A *fault-containment region* (FCR) is a region in a system designed to ensure faults do not propagate to other regions. The easiest way to ensure this is to physically isolate one FCR from another. However, FCRs may need to communicate, hence they share channels. An FCR containing a monitor may need to share a channel with the SUO. Care must be taken to ensure faults cannot propagate over these channels. In the case of ultra-reliable systems, the only way to achieve the level of fault tolerance demanded of the system is by hardware replication that demands complex hardware redundancy management software.

Challenge: Isolating failures so that RV should not be rendered inoperable by the same failure conditions that impact the SUO.

Copilot Approach: Fault-tolerant RV has been an ongoing topic of investigation for the Copilot research group. The avionics industry has been migrating away from federated systems toward the use of integrated modular avionics that provide fault tolerance as a service. The Aeronautical Radio, Incorporated (ARINC) 653 [4] compliant RTOS provides temporal and spatial partitioning guarantees so applications can safely share resources. The Copilot group has been investigating design patterns for implementing fault-tolerant RV on such platforms [11]. Monitors are run on the same nodes as the software being monitored, but in separate partitions. Monitoring tasks executing in a separate partition observe the state of the executing program through very restricted channels that preserve the isolation guarantees. The spacial and temporal protections provided by ARINC 653 keep the monitors safe from other programs running on the same system.

Systems that need to be ultrareliable typically must employ redundancy to tolerate the most pernicious faults such as a *Byzantine fault* (i.e., a fault in which different nodes interpret a single broadcast message differently). There have been documented incidents in critical avionics where sensors failed in a Byzantine fashion with the potential to affect a vehicle's safety. The aircraft horizontal separation criteria in Section 3 depends on reliably sensing the position and velocity of both systems. In earlier work [28], we have addressed this issue in a case study where a system had redundant sensors and Copilot monitors performed Byzantine exchange and majority voting to create a system that could tolerate a single Byzantine fault. Fault injection testing was performed along with flight tests. The hardware used in these experiments were commodity microprocessors, but we have recently bought Mil-Spec hardened processors that are more reliable when operating under varying environmental conditions.

7 Do No Harm

The RV components must be composed with the SUO so that they are executing in parallel with the SUO. Care must be taken that the RV system itself does not compromise the correct functioning of the SUO. For instance executing monitors may impact timing and scheduling. Care must also be taken that any instrumentation of the SUO does not affect the functional correctness. In large systems, there are likely to be many monitors running; each monitor might trigger different steering procedures. A common pattern when things go wrong in complex safety-critical systems is that many alerts are sounded simultaneously often placing a burden on a human operator to sort things out. One can easily envision an analogous situation where several monitors detect violations, triggering their respective steering procedures. Hence it is necessary to verify that these different steering procedures do not interact with each other in ways that could compromise safety. In summary, high-assurance RV must uphold the Hippocratic oath "to do no harm". Ideally, we would formulate a noninterference theorem and the RV framework would produce a proof certificate that the composed system satisfies the property.

Challenge: Assured RV must safely compose with the SUO.

Copilot Approach: The Copilot research group has yet to develop a general theory of RV noninterference, but we have made a number of design decisions with this in mind. For instance, the choice of monitoring system state through sampling was a deliberate attempt to minimize interference with the SUO. Running monitors in separate partitions on an ARINC 653 compliant RTOS as discussed in Section 6 ensures that any fault in the RV will not negatively affect the executing SUO. The RTOS scheduler also provides guarantees that a missed deadline in the RV does not affect the SUO.

8 Monitor Specification Correctness

As RV is applied to guarantee sophisticated properties, the monitor specifications themselves will grow in complexity and may become prone to error. Our experience with a number of case studies involving complex monitor properties is that we were able to discover many simple theorems that should hold for a correct specification. Hence, applying formal proof tools to the monitors to ensure they are correct can safeguard that the last line of defense is actually effective. Ideally, specification verification capabilities should be integrated into the RV framework so engineers could write specifications, verify their correctness, and generate monitors in a seamless fashion.

Challenge: Assure the correctness of the monitor specification.

Copilot Approach: Copilot supports automated proofs of specification properties through its `Copilot.Theorem` module. Applying a "synchronous observer" approach [17], properties *about* Copilot programs are specified *within* Copilot itself. In particular, properties are encoded with standard Boolean streams and Copilot streams are sufficient to encode past-time linear temporal logic [18].

A proposition is a Copilot value of type `Prop Existential` or `Prop Universal`, which can be introduced using `exists` and `forall`, respectively. These are functions taking as an argument a normal Copilot stream of type `Stream Bool`. Propositions can be added to a specification using the `prop` and `theorem` functions, where `theorem` must also be passed a tactic for automatically proving the proposition. Currently, proof engines based on Satisfiability Modulo Theories (SMT) are used to discharge proofs. The Copilot prover was first introduced in [16], where its utility was demonstrated in assuring notoriously subtle voting algorithms.

In the course of the analysis of the separation criteria, a team of domain experts used the PVS interactive prover to prove theorems that characterize the correctness of the criteria. We were able to apply the Copilot prover using Z3 [12] to prove many of these theorems within the Copilot framework. Among the properties proven about the horizontal separation criteria are:

$$\begin{aligned}
&\texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon, \boldsymbol{vx}, \boldsymbol{vy}) \iff \texttt{horiz\_criteria}(-\boldsymbol{sx}, -\boldsymbol{sy}, \epsilon, -\boldsymbol{vx}, -\boldsymbol{vy})\\
&(\texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon, \boldsymbol{vx}, \boldsymbol{vy}) \wedge\\
&\quad \texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon, \boldsymbol{wx}, \boldsymbol{wy})) \implies \texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon, \boldsymbol{vx}, \boldsymbol{vy})\\
&(\texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon, \boldsymbol{vx}, \boldsymbol{vy}) \wedge\\
&\quad \texttt{horiz\_criteria}(\boldsymbol{sx}, \boldsymbol{sy}, \epsilon', \boldsymbol{vx}, \boldsymbol{vy})) \implies \epsilon = \epsilon'
\end{aligned}$$

A few of the properties proven in PVS involve continuous mathematics that remains beyond the capabilities of fully automated tools, but combined with testing, we have a convincing argument that the specification is correct.

9 Correct monitors

RV frameworks apply sophisticated algorithms to synthesize monitors from specifications. In safety-critical systems, subtle errors in the translation process can have potentially catastrophic consequences and consequently, a safety case for assured RV must include evidence of the correctness of the translation process.

Challenge: There should be assurance arguments with evidence that executable monitors correctly implement the specification. The monitor implementation should not be susceptible to unsafe or undefined behaviors such as buffer and floating point overflows.

Copilot Approach: The small Copilot interpreter can be seen as providing an executable operational semantics for the Copilot language. As reported in [27], our first efforts in monitor synthesis assurance were to support regression tests for the semantics of the EDSL using Haskell's QuickCheck [9] property testing engine. Type-correct Copilot programs get randomly generated and output from the interpreter is compared against the actual monitor. QuickCheck testing uncovered a number of bugs during early development of the monitor synthesis software. Among those bugs caught were forgotten witnesses needed by the code generation tools. The testing also highlighted differences in how GCC and Haskell implemented floating point numbers, without either violating the IEEE floating point standard.

Recent work leverages light-weight verification tools for monitor synthesis assurance. The process of translating a specification into a monitor transforms an abstract syntax tree (AST) of the "core" language representation into a SBV AST. SBV's C code generation capabilities are used to generate executable C code. To facilitate monitor verification, Copilot produces Hoare-logic style contracts directly from the Copilot core representation independent of the monitor generation process. The contracts are written in the ANSI C Specification Language (ACSL) [5], an assertion language for specifying behavioral properties of C programs in first-order logic. Each file has a contract with an ACSL postcondition specifying the subexpression of the core AST representation that the function should implement. Frama-C's [14] WP deductive verification engine is employed to prove that the code does indeed satisfy the contract. Deductive verification tools have evolved quite a bit recently, but scalability is still an issue. However, the verification is tractable because the translation process creates separate C functions for subexpressions of a large expression.

An example of an annotated monitor C function follows:

```
/*@
 assigns \ nothing;
 ensures \ result == (((ext_ident_double_8) - (((ext_minimal_horizontal_separation) *
                                  (ext_minimal_horizontal_separation)))));
*/
SDouble ext_sqrt_9_arg0(const SDouble ext_ident_double_8,
       const SDouble ext_ownship_position_x, const SDouble ext_intruder_position_x,
       const SDouble ext_ownship_position_y,  const SDouble ext_intruder_position_y,
       const SDouble ext_minimal_horizontal_separation)
```

```
{   const SDouble s0 = ext_ident_double_8;
    const SDouble s5 = ext_minimal_horizontal_separation;
    const SDouble s6 = s5 * s5;
    const SDouble s7 = s0 - s6;
    return s7;  }
```

Frama-C's WP plugin easily proves that the function satisfies the contract.

While this analysis demonstrates a faithful translation from core language to C code, it elides the issues that arise performing floating point arithmetic. We applied both the RV-Match tool [2] and Frama-C's abstract interpretation value analysis plugin to detect when floating point arithmetic produces infinite values or not a number (NaN). The RV-Match C undefinedness checker found a divide-by-zero error due to our initializing a variable to zero. The abstract interpreter produced warnings for every floating point operation. In the case of the `ext_sqrt_9_arg0` function, the value analysis produces the following warnings:

```
\ext_sqrt_9_arg0.c:41:[kernel] warning: non-finite double value
([-1.79769313486e+308 .. 1.79769313486e+308]): assert
\is_finite((double)(s5*s5)); ext_sqrt_9_arg0.c:42:[kernel] warning:
non-finite double value ([-1.79769313486e+308
.. 1.79769313486e+308]): assert \is_finite((double)(s0-s6));
ext_sqrt_9_arg0.c:30:[value] Function ext_sqrt_9_arg0: postcondition
got status unknown.
```

Applying domain specific knowledge about the bounds on the velocity and state vectors eliminated this warning. At present, we must add these bounds to the contracts by hand, but intend to generate such information during monitor generation.

Assurance All The Way Down: Having assured that the C code implementing the monitor is correct, how can we guarantee that the executable binary code correctly implements the C program? For Copilot, we have experimented with using the verified Compcert compiler [24] to generate binaries. Unfortunately, Compcert does not yet target many of the processors used in our experiments limiting its utility.

10 Additional Challenges

The presentation so far has examined challenges in assured RV that have been addressed in our research. In this section, we will raise three of the key additional challenges that we have identified as critical to address in future work.

Safe Steering: The problem of what to do when a specification has been violated is one of the most thorny problems in high-assurance RV and almost completely application dependent. The simplest action is to log the violation for further analysis or raise an alarm for humans to intervene, but in many cases, the RV system must take proactive steps to preserve safety. For an autonomous robot, putting the system into a quiescent state may be a safe default operation depending on the operating environment. In the case of an adaptive control system, the RV framework may switch to a conventional controller, but whether this re-establishes safety depends on many factors. In many domains, the challenge in constructing an assured safe steering algorithm may be as difficult as constructing the adaptive autonomous algorithm itself.

Challenge: Verify the correctness and safety of the steering performed from any viable system state once a specification violation is detected.

Predictive Monitors: Applying RV to application domains that have strict timing constraints, such as an adaptive control system, raises many technical challenges. It is imperative that the monitor detects that the adaptive controller is about to lose stability in time to switch to a safe controller. In the case of our running example, the RV system needs to detect that two aircraft are about to lose separation in time for them to take corrective action. Assured predictive monitors are needed, but much work remains to be done. Johnson et. al. [19] is a promising approach to predictive monitoring for controllers, but the general problem is very domain specific. Assured predictive monitoring remains a research challenge for the RV community.

Challenge: The monitor should detect impending violations of the specification and invoke the safety controller in time to preserve safe operation.

Secured RV: Adding more software or hardware to a system has the potential to introduce vulnerabilities that that can be exploited by an attacker. Copilot and many other RV frameworks generate monitors that have constant time and constant space execution footprints and while this eliminates some common attacks, it does not provide general protection. Every sensor and unauthenticated message may contribute to the attack surface. If an attacker can trick the systems into a monitor specification violation, the triggered steering behavior may itself constitute a denial-of-service attack. Future research is needed in identifying attack surfaces and suitable techniques to thwart adversaries from turning the RV meant to protect a system into a liability that exposes the system to attack.

Challenge: Assured RV should not introduce security vulnerabilities into a system.

11 Better Together

High-assurance RV will only become practical if there is accompanying integrated tool support for verification and validation. From the perspective of a researcher in high-assurance RV, engaging with the communities building static analysis tools and proof engines seems obvious, especially in light of the fact that regulatory bodies that govern many safety-critical systems are increasingly willing to accept the analysis produced from such tools as evidence that can be applied to certification [29]. Similarly, there are many features of RV that make it a great target for the tool builder. For instance, there are formal specifications to work with and monitor code is generally small and conforms to coding practices that are friendly to static analysis.

Several of the challenges we have raised for high-assurance RV are really at the system level. Tools that can assist domain experts in validating safety properties are sorely needed. As in our case study, the safety properties of cyber-physical systems involve continuous mathematics. While advances in SMT solvers have been impressive, it is still necessary to often resort to using an interactive theorem prover. There are many opportunities to design domain specific decision procedures that would increase the utility of automated proof tools.

The problem of RV observability provides a rich source of problems for tool builders. If the RV approach involves instrumenting code, then static analysis can both assist in the instrumentation and prove that the instrumentation did not affect the correctness of the code. If sampling, static analysis has the potential to inform when to sample.

Floating point arithmetic is a source of problems that is readily amenable to static analysis and proof. In our work, we applied abstract interpretation to monitor source code, but analysis could be done at the specification level with proof obligations flowing down to the monitor code.

Applying tools to verify monitor correctness makes so much sense that it should be de rigueur. In Copilot, we applied deductive verification to verify the correctness of the translation from specification to monitor code. Monitors have many characteristics that make automatic proofs tractable, but the monitor synthesis must generate code suitable for the tool being used.

12 Conclusion

High-assurance RV has the potential of becoming the avenue to assuring otherwise unassurable IA safety-critical systems. We have presented a number of challenges that we have identified as barriers to actualizing high-assurance RV and surveyed how we have addressed these challenges in the course of our research using the Copilot framework. We hope this list will be useful to RV researchers as they apply their own work to safety-critical systems. In addition, we believe we have demonstrated the efficacy of applying light-weight formal methods tools to address many of these challenges. Progress on these issues is likely to come faster if a multidisciplinary approach is taken with domain specialists, safety engineers and verification tool builders collaborating with RV researchers. Much work remains and the list of challenges is likely to grow even as researchers solve many of the issues raised.

Acknowledgements: The Copilot project has been conducted in collaboration with Dr. Lee Pike (Galois). Jonathan Laurent (ENS Paris) and Chris Hathhorn (University of Missouri) did most of the coding of the `Copilot.Theorem`. Georges-Axel Jaloyan recently added the new monitor verification capabilities.

References

1. The DOT language. `http://www.graphviz.org/content/dot-language`.
2. RV-Match. `http://runtimeverification.com/match/`.
3. SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, 1996.
4. Incorporated (ARINC) Aeronautical Radio. Avionics application software standard interface: Part i - required services (arinc specification 653-2), 2015.
5. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.10*, 2015.
6. Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *17th Intl. Symposium on Formal Methods (FM)*, 2011.
7. Ricky W. Butler. A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center, 2008.
8. F. Chen and G. Roşu. Java-MOP: a monitoring oriented programming environment for Java. In *11th Intl. Conf. on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.
9. Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.
10. National Research Council. *Autonomy Research for Civil Aviation: Toward a New Era of Flight*. The National Academies Press, 2014.
11. Kaveh Darafsheh. Runtime monitoring on hard real-time operating systems. Master's thesis, East Carolina University, 2015.
12. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
13. S. Fischmeister and Y. Ba. Sampling-based program execution monitoring. In *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 133–142, 2010.
14. Frama-C. Accessed March, 2016. `http://frama-c.com/index.html`.
15. Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.
16. Jonathan Laurent Alwyn Goodloe and Lee Pike. Assuring the guardians. In *Proceedings of the 6th Intl. Conference on Runtime Verification*, LNCS 9333. Springer, September 2015.
17. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*. Springer Verlag, June 1993.
18. Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.
19. Taylor T. Johnson, Stanley Bak, Marco Caccamo, and Lui Sha. Real-time reachability for verified simplex design. *ACM Transactions on Embedded Computing Systems*, September 2015.

20. Aaron Kane. *Runtime Monitoring for Safety-Critical Embedded Systems.* PhD thesis, Carnegie Mellon University, 2015.
21. David A. Karr, Robert A. Vivona, David Roscoe, Stephen M. DePascale, and Maria Consiglio. Experimental performance of a genetic algorithm for airborne strategic conflict resolution. In *Proceedings of AIAA Guidance, Navigation and Control Conference*, 2008.
22. M. Kim, I. Lee, S. Kannan, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. *Formal Methods in System Design*, 24(1):129–155, 2004.
23. John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550. ACM, 2002.
24. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52:107–115, July 2009.
25. Anthony Narkawicz and César Muñoz. State-based implicit coordination and applications. Technical Publication NASA/TP-2011-217067, NASA, Langley Research Center, Hampton VA 23681-2199, USA, March 2011.
26. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010.
27. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP).* ACM, September 2012.
28. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4), 2013.
29. Formal methods supplement to do-178c and do-278a. RTCA, Inc., 2011. RCTA/DO333.
30. RTCA. Software considerations in airborne systems and equipment certification. RTCA, Inc., 2011. RCTA/DO-178C.
31. John Rushby. Runtime certification. In *Eighth Workshop on Runtime Verification (RV08)*, volume 5289 of *LNCS*, pages 21–35, 2008.
32. John Rushby. A safety-case approach for certifying adaptive systems. In *AIAA Infotech*, 2009.
33. Lui Sha. Using simplicity to control complexity. *IEEE Software*, pages 20–28, July/August 2001.