Automated SysML v2 System Model to Memory-Safe Language Code Generation for Avionics Applications

David Hardin, Isaac Amundson, Junaid Babar, Darren Cofer, Saqib Hasan, and Karl Hoech
Applied Research and Technology
Collins Aerospace
USA
first.last@collins.com

Jason Belt, John Hatcliff, and Robby
Department of Computer Science
Kansas State University
USA
last@ksu.edu

Stefan Hallerstede
Department of Electrical and Computer Engineering
Aarhus University
Denmark
sha@ece.au.dk

***Abstract*—One of the greatest challenges of Model-Based Systems Engineering (MBSE) for Digital Avionics applications is ensuring that the system model and design/implementation remain "in-sync" during product development. We are creating a revolutionary MBSE environment that allows non-specialist developers to specify models in the SysML v2 systems modeling language, automatically generate skeletal implementations of those models in a memory-safe language (MSL), specifically Rust, as well as state and prove formal properties about the system model that can be refined and reproved against the generated design. These systems can be selected to be hosted on either Linux or the verified seL4 microkernel, or simulated using a Java Virtual Machine (JVM) based environment. In this paper, we will present our SysML v2-based toolchain, demonstrate its code generation capability on a simple digital avionics system example, and demonstrate its automatic property specification and proof capability, all in the context of an industrial Continuous Integration/Continuous Deployment (CI/CD) framework.**

***Index Terms*—Model-Based Systems Engineering, Formal Verification, Memory-Safe Languages, Rust Programming Language**

I. Introduction

A significant challenge in Model-Based Systems Engineering (MBSE) for Digital Avionics applications is providing allocation to/traceability from the corresponding software and hardware design, particularly ensuring that the system model and design/implementation remain "in-sync" during product development, as well as guaranteeing that assurance properties established at the system model level hold at the design/implementation level. This challenge is heightened due to the fact that modern avionics systems are sophisticated real-time cyber-physical systems that are often exposed to a wide variety of cyber threats. Furthermore, aerospace systems are subject to intense regulatory scrutiny due to the certification requirements of this domain.

Without the capability to provide allocation/traceability between systems models and implementations, safety and security flaws are often only discovered during testing late in the development process. Worse yet, they may be discovered only after the product has been fielded, necessitating extremely expensive and time-consuming remediation. Recognizing this challenge, our team is developing a revolutionary MBSE tool environment under the DARPA Pipelined Reasoning of Verifiers Enabling Robust Systems (PROVERS) program. The stated goal of PROVERS is "to make formal methods accessible to non-experts (e.g., traditional software developers and systems engineers) while minimizing the impact on their existing processes and performance." [1]

Our PROVERS toolchain, named INSPECTA (Industrial-Scale Proof Engineering for Critical Trustworthy Applications) integrates design, code generation, and formal verification activities, enabling systems engineers to design-in resiliency for today's complex avionics systems using the SysML v2 systems modeling language. Our tools automatically generate skeletal implementations for SysML v2 models in a memory-safe language (MSL), specifically Rust, allowing developers to state and prove formal properties about the system model that can be automatically refined and reproved against the generated design in Rust. This capability to automatically generate verification-enhanced memory-safe language code from architecture models featuring assume/guarantee contracts is novel, providing high assurance while saving engineering time and effort.

A fundamental aspect of our approach is the use of system architecture models to provide a framework for analyzing system behavior and organizing the assurance evidence produced. Architecture modeling allows engineers to describe the important elements of distributed, real-time, embedded systems (processors, memory, buses, processes, threads, and data interconnections) with sufficiently rigorous semantics that can support formal reasoning. SysML is one such general-purpose modeling language developed to facilitate MBSE. SysML can be used to represent system requirements, structure, and behavior, as well as specify properties that can be used to analyze and verify the system. SysML v2, the most recent version of the standard, includes a number of improvements which we are leveraging to add compositional assume/guarantee contracts to SysML v2 to enable developers to state and prove system-level properties, much as we have previously accomplished [2] for the Architecture Analysis and Design Language (AADL) [3].

We have developed a unique multi-platform, multi-language skeletal code generation framework from system models, called HAMR. HAMR can be used to automatically generate code for deployment for the Java Virtual Machine (primarily for host-based simulation/testing), Linux, or the verified seL4 microkernel. seL4 is formally verified from its high-level security properties down to its binary implementation [4]. Notably, seL4 features mathematically proven separation between threads, thus providing a very high assurance foundation upon which to produce application-level property proofs. We are currently targeting code generation for MSLs, including precondition/postcondition annotations necessary for formal verification. These annotations are refined from the system-level contracts defined in the SysML v2 model. MSLs avoid many common vulnerabilities that are the bane of C/C++ development, and are increasingly required for high-assurance development, in the wake of recommendations by cybersecurity agencies of the U.S. Government. The ability to automatically generate MSL code from a system model helps to produce complete systems, even when MSL developers are scarce.

Recent HAMR development has targeted the Rust programming language. Rust is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. Additionally, HAMR supports a precondition/postcondition specification and verification tool for Rust called Verus, under development by a team of researchers led by Carnegie Mellon University.

In this paper, we will present our SysML v2-based toolchain, demonstrate its Rust code generation capability on a simple digital avionics system example, and demonstrate its property specification and proof capability, all in the context of a Continuous Integration/Continuous Deployment (CI/CD) framework of the type used by avionics developers.

II. The SysML v2 Systems Modeling Language

SysML v2 is the second major version of the Systems Modeling Language, a standard language for modeling systems and their behavior [5] promulgated by the Object Management Group. It builds upon SysML v1, offering enhanced features such as a textual representation, standard API, and improved expressiveness for modeling complex systems, as well as a formal semantics. SysML v1 has been adopted by industry, but mostly has been used for requirements elicitation and documentation support. However, the increased expressiveness of SysML v2, coupled with the ability to interchange models with third party tools via its textual representation, enables a much deeper integration of architectural modeling with detailed development, as will be discussed in more detail in Section IV.

Note that we process a subset of SysMLv2 where the modeling elements have a semantic correspondence to AADL modeling elements [6] for which we have developed a formal semantics [7]. This enables our previous AADL-based tooling to be readily transitioned into SysMLv2, promising a broader reach and more extensive tooling. This also tracks a larger effort within the Object Management Group to provide standardized AADL-like semantics for SysML v2 systems.

III. Assume/Guarantee Contract Verification for System Models

Our research team has pioneered the use of compositional assume/guarantee reasoning for MBSE [2], [8]. We have developed the Assume Guarantee Reasoning Environment (AGREE), a compositional, assume-guarantee-style model checker for system architecture models [9]. AGREE proves properties about one layer of the architecture using properties allocated to subcomponents. The composition is performed in terms of assumptions and guarantees that are provided for each component. Assumptions describe the expectations the component has on the environment, while guarantees describe bounds on the behavior of the component.

AGREE uses k-induction as the underlying algorithm for model checking. System models and AGREE contracts are first translated into the Lustre language [10], including verification conditions for consistency and correctness of the contracts. The model checker then attempts to find any model execution traces that would violate these conditions using one of several Satisfiability Modulo Theories (SMT) solvers. If the model checker covers all reachable states in the model without finding a violation then the properties are proved.

More recently, a team led by Kansas State University researchers have developed another contract language for system architecture models, called GUMBO [11]. GUMBO differs from AGREE in that it is not based on Lustre dataflow primitives, and better supports code level checking. We are currently working with KSU on a harmonization of AGREE and GUMBO for use with SysML v2 models as part of the DARPA PROVERS program.

A SysML v2 model for a temperature regulator component with GUMBO assume/guarantee contracts is depicted in Fig. 1. This kind of component can be found in, for example, pitot tube heaters, vane heaters for Angle-of-Attack (AoA) sensors, or battery management systems. In this case, the GUMBO contracts are introduced into the model via the SysML v2 `language` construct.

Once the system architecture has been modeled and the component behavior is specified using assume-guarantee contracts, we can then use a fully-automatic model checker to verify the consistency of these contracts.

1) Component interfaces – The output guarantees of each component must be strong enough to satisfy the input assumptions of downstream components.
2) Correctness of implementations – The input assumptions of a system along with the output guarantees of its *sub*-components must be strong enough to satisfy its output guarantees.

This hierarchical strategy for reasoning about contracts, or *compositional analysis*, reduces the computational complexity of system verification by breaking down the larger problem into more manageable fragments. Verification of a system does not directly depend on the implementations of its components (or their sub-components), but only on their contracts.

IV. HAMR: Skeletal Code Generation from System Models

The High Assurance Modeling and Rapid engineering for embedded systems tool (HAMR) [12] is a multi-platform, multi-language code generation framework for system architecture models. HAMR is often used to generate code for deployment for the seL4 microkernel, but system and component prototyping is also supported utilizing HAMR's code generation capability for the Java Virtual Machine (JVM) and Linux, as depicted in Fig. 2.

Using seL4 as a foundation, HAMR enables model-based development and systems engineering frameworks to be used for seL4-based applications. One of the primary objectives is to support system builds that leverage seL4 separation and information flow guarantees to achieve the system-model-specified component isolation and inter-component communication needed for cyber-resiliency. HAMR ensures that seL4 is configured to permit the exact inter-component information flows analyzed and visualized at the model level.

For each system model thread component, HAMR generates a thread code skeleton and application programming interfaces (APIs) for communicating over the ports declared on the component. For components that are implemented manually, the developer fills out the thread skeleton with detailed application code. HAMR supports coding component application logic in either C, Slang [13] (a high-assurance memory-safe subset of Scala), and now Rust.

HAMR generates component infrastructure and integration code implementing the semantics of thread scheduling, thread dispatching, and port-based communication defined by AADL or SysML v2. For port communication, shared memory communication (data ports), buffered messaging (event data ports), and buffered notification (event ports) are supported. HAMR code generation is staged using a translation architecture that facilitates adding new backends for different target platforms. Almost all the infrastructure code is implemented in Slang, which can then be used for JVM deployments or translated to C for Linux or seL4 deployments.

HAMR is istelf implemented in Slang, and produces a Slang-based implementation of the run-time framework, which can be viewed as a high-level reference implementation of architecture model semantics in a high-level memory-safe language. Automatic translation ("transpilation") of this reference implementation in Slang to Rust or C on different platforms helps establish semantic consistency across those platforms. This reference implementation can also be easily compiled to Java Virtual Machine bytecodes, thus directly supporting a JVM-based simulation environment.

The seL4 deployment utilizes the Microkit [14] microkernel-based embedded systems code-generation framework to configure the microkernel. A Microkit system is built from a set of individual programs that are isolated from each other, as well as the underlying kernel, in *protection domains*. Protection domains can interact by calling *protected procedures* or sending *notifications*. The HAMR-generated Microkit configuration directly encodes the system model's component and communication topology and includes a standardized run-time infrastructure supporting thread scheduling, inter-thread communications, etc. We incorporate the previously-developed (and proved) seL4 domain scheduler into the Microkit codebase in order to enforce time partitioning and provide static cyclic scheduling. We will be able to use the more advanced Mixed-Criticality Scheduler (MCS) for seL4 [15] when its proofs of correctness down to the binary level are complete (this work is currently in progress as part of the DARPA PROVERS program).

HAMR also supports Linux-based virtual machine components in the seL4 deployment and the ability to run the entire system on the QEMU emulator. HAMR automatically configures virtual machine based components, and this feature is used to sandbox the untrusted legacy code for the Mission Planner in the example UAV system. The QEMU emulator support facilitates rapid prototyping for test, debug, and analysis, and it enables automated regression testing.

As part of its code generation process, HAMR produces flow graphs reflecting the inter-component information flow at both the architecture level and the Microkit level for the seL4 deployment. Visual representations are provided for manual inspection, and SMT-based representations are generated for formal reasoning.

The relationships amongst the various layers of the system design, the tools that allow one to proceed from the more abstract layers to the more concrete layers, as well as the proof artifacts that can be generated at each layer are depicted in Fig. 3. One type of proof that we do not address at length

```
package Regulate {

  part def Manage_Heat_Source_i :> Thread {
    attribute :>> Dispatch_Protocol = Supported_Dispatch_Protocols::Periodic;
    attribute :>> Period = 1000 [millisecond];
    attribute Domain: CASE_Scheduling::Domain = 9;
    attribute Microkit_Language: HAMR::Microkit_Language = HAMR::Microkit_Languages::Rust;

    // ======== INPUTS =======
    // current temperature (from temp sensor)
    in port current_tempWstatus : DataPort { in :> type : Data_Model::TempWstatus_i; }
    // lowest and upper bound of desired temperature range
    in port lower_desired_temp : DataPort { in :> type : Data_Model::Temp_i; }
    in port upper_desired_temp : DataPort { in :> type : Data_Model::Temp_i; }
    // subsystem mode
    in port regulator_mode : DataPort { in :> type : Data_Model::Regulator_Mode; }

    // ======== OUTPUTS =======
    // command to turn heater on/off (actuation command)
    out port heat_control : DataPort { out :> type : Data_Model::On_Off; }

    language "GUMBO" /*{
      state
        lastCmd: Data_Model::On_Off;

      initialize
         guarantee
           initlastCmd: lastCmd == Data_Model::On_Off.Off;
         guarantee REQ_MHS_1 "If the Regulator Mode is INIT, the Heat Control shall be set to Off."
           heat_control == Data_Model::On_Off.Off;

      compute
        // assumption on set points enforced within the Operator Interface
        assume lower_is_lower_temp: lower_desired_temp.degrees <= upper_desired_temp.degrees;

        // the lastCmd state variable is always equal to the value of the heat_control output port
        guarantee lastCmd "Set lastCmd to value of output Cmd port":
          lastCmd == heat_control;

        compute_cases
          case REQ_MHS_1 "If the Regulator Mode is INIT, the Heat Control shall be set to Off.":

            assume regulator_mode == Data_Model::Regulator_Mode.Init_Regulator_Mode;
            guarantee heat_control == Data_Model::On_Off.Off;

          case REQ_MHS_2 "If the Regulator Mode is NORMAL and the Current Temperature is less than
                          the Lower Desired Temperature, the Heat Control shall be set to On.":

            assume (regulator_mode == Data_Model::Regulator_Mode.Normal_Regulator_Mode) &
                     (current_tempWstatus.degrees < lower_desired_temp.degrees);
            guarantee heat_control == Data_Model::On_Off.Onn;
    [...] } */
  [...] }
}
```

Fig. 1. SysML v2 model with GUMBO contracts.

in this paper, but which provides extremely high assurance, is verified synthesis directly from specifications in a formal logic to embedded source code. We have employed this technique on the current effort, for example, in order to synthesize high-assurance remote attestation protocols [16]. Additionally, the source code-to-binary correspondence proofs depicted in Fig. 3 are generally only performed for the seL4 operating system proofs [4].

V. Rust, A Memory-Safe Programming Language for Critical Systems

Memory-safe programming languages have garnered significant interest in recent years, with memory-safe language initiatives issuing from NSA [17], as well as the White House Office of the National Cyber Director [18], and memory-safe language requirements starting to appear in U.S. Government contracting. One of the advantages of memory-safe languages is the capability to reason about application code written in the imperative style favored by industry, but without

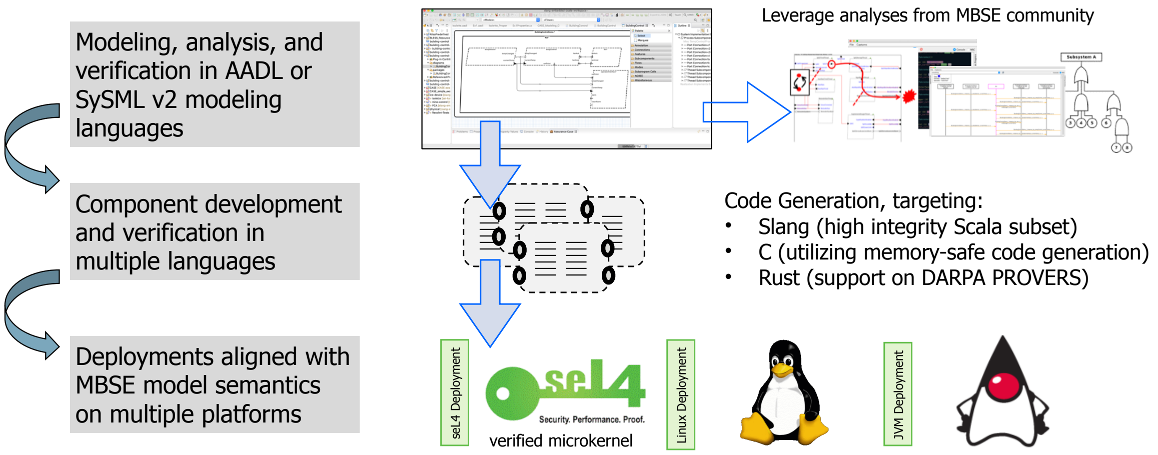

Modeling, analysis, and verification in AADL or SysML v2 modeling languages
Component development and verification in multiple languages
Deployments aligned with MBSE model semantics on multiple platforms
Leverage analyses from MBSE community
Code Generation, targeting:
• Slang (high integrity Scala subset)
• C (utilizing memory-safe code generation)
• Rust (support on DARPA PROVERS)
seL4 Deployment
seL4 Security. Performance. Proof.
verified microkernel
Linux Deployment
JVM Deployment

Fig. 2. HAMR overview.

the verification snarls of the unrestricted pointers of C/C++. Much progress has been made to this end in recent years, and developers can now verify the correctness of common algorithm and data structure code that utilizes common idioms such as records, loops, modular integers, and the like, and verified compilers can guarantee that such code is compiled correctly to binary [19]. Many open-source verification tools have emerged to reason about code written in memory-safe languages (see Section V-A for details).

The Rust programming language has attracted particular attention, due to its unique memory ownership model and efficient compiler, capable of producing code that is competitive in efficiency with C/C++ compilers. Google [20] and Amazon [21] are major Rust adopters, and Rust is now being used in Linux kernel development [22]. After spending decades dealing with a never-ending parade of security vulnerabilities due to C/C++, which continue to manifest at a high rate [23] despite their use of sophisticated C/C++ analysis tools, Microsoft announced at its BlueHat 2023 developer conference that it was beginning to rewrite core Windows libraries in Rust [24]. A distinguishing feature of Rust is that objects may only have one owner. This means, for example, that it is not possible to return a pointer to a local variable from a function to its enclosing scope. The Rust compiler produces code that is competitive in speed to C/C++ compilers, including code to perform array bounds checking, as well as arithmetic overflow checking.

For formal methods researchers, Rust presents the opportunity to reason about application-level logic written in the imperative style favored by industry, but without the snarls of the unrestricted pointers of C/C++. Much progress has been made to this end in recent years, and developers now have access to tools that are capable of automatically verifying the correctness of Rust algorithm and data structure code that utilizes common Rust idioms such as records, match statements, while loops, modular integers. Particular progress has been made in the area of hardware/software co-design algorithms, where array-backed data structures are common [25], [26].

A. *Rust Verification with Verus*

A team led by Carnegie Mellon University is developing Verus, an SMT-based tool for formally verifying Rust programs [27]. With Verus, programmers express proofs and specifications using extended Rust syntax (introduced using Rust's macro facility), allowing proofs to take advantage of Rust's high-assurance ownership restrictions. This extended verificaion syntax is "erased" for code compilation, allowing Verus-enhanced code to be compiled using the standard Rust compiler. We are working with the Verus developers on the DARPA PROVERS program to create a verification environment for Rust that developers who are not formal methods specialists can use effectively.

Like many verification-enhanced programming environments, Verus provides means to express formal preconditions for a given function, which provide the needed input constraints to achieve a successful verification for that function (expressed as `requires` blocks), as well as postconditions that provide the output conditions (expressed as `ensures` blocks) that can be proved to arise from the preconditions. Verus also provides means to express loop invariants (via `invariant` blocks within the loop bodies).

Verus-annotated Rust code has been automatically generated by HAMR from the temperature regulator SysML v2 model of Fig. 1 is given in Fig. 4. Verus automatically verifies this generated code, providing developers with assurance that their system-level contracts continue to hold at the code level. Note that AGREE/GUMBO `assume` statements map to Verus `requires`, and that AGREE/GUMBO `guarantee` maps to Verus `ensures`.

VI. Continuous Integration/Continuous Deployment Support for Avionics

One of the main usability goals of PROVERS is to integrate formal methods tools "into a development pipeline enabling

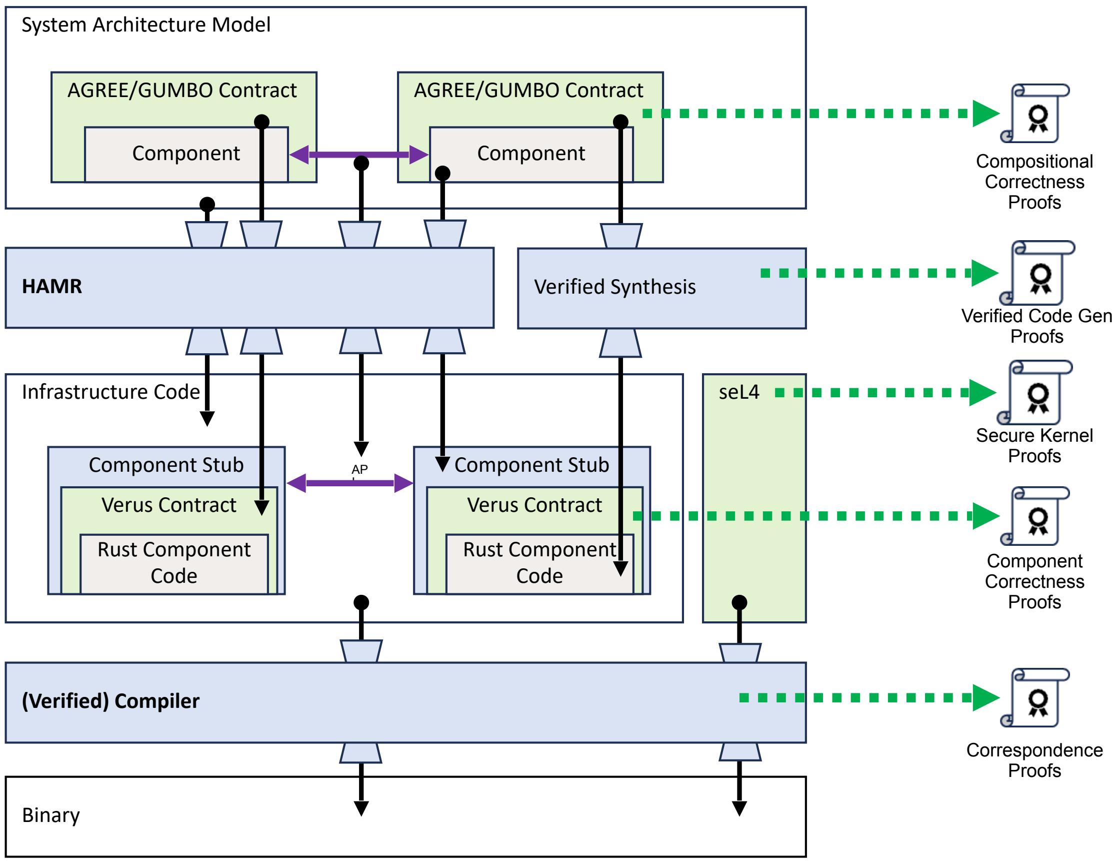


Fig. 3. High-Assurance System Development Layers, Tools, and Proof Artifacts.

a continuous flow of capabilities over time while maintaining high levels of assurance.” [1] Thus, we are working to integrate our formal-methods-based toolchain into a Continuous Integration/Continuous Deployment (CI/CD) pipeline currently in use at Collins Aerospace. In keeping with the “DevSecOps” trend for high-assurance development, we introduce “ProofOps” into the pipeline. Our goal is to invoke certain formal methods tools on every commit, thus requiring that these tools be performant, and provide meaningful feedback to the development engineers. Thus, we are developing a ProofOps “dashboard” that provides “at a glance” status for the proof-based tool operations. A prototype dashboard is depicted in Fig. 5.

VII. Related Work

Compositional reasoning for system architecture models was brought to prominence on the DARPA High-Assurance Cyber Military Systems (HACMS) program [8]. HACMS researchers utilized AADL modeling, AGREE contracts, and the formally-verified seL4 kernel to produce “clean sheet” designs and implementations, including the Boeing Little Bird autonomous helicopter platform, that successfully resisted cyber attack by a dedicated “red team”, as well by hackers at the DEF CON conference. The DARPA Cyber Assured System Engineering (CASE) extended the tools and techniques pioneered on HACMS to support systems that include significant “legacy” code that cannot easily be redesigned. The CASE team pioneered the development of *security-enhancing architectural transformations* to improve the security posture of existing systems by the addition of provably-correct filters, monitors, and remote attestation components [2].

A number of verification-enhanced memory-safe language environments are currently in use for critical system development. SPARK 2014 [28] is a contract-based specification and verification framework for a safety-critical subset of Ada. SPARK verification tools translate SPARK programs into the Why3 verification framework [29] to prove that SPARK programs conform to program contracts. Formal verification systems for Rust include Creusot [30], based on WhyML; Prusti [31], based on the Viper verification toolchain; and RustHorn [32], based on constrained Horn clauses. Amazon Web Services is developing a model-checker for Rust, Kani [33].

VIII. Conclusion

We have presented a high-assurance system development toolchain for SysML v2 targeting a memory-safe program-

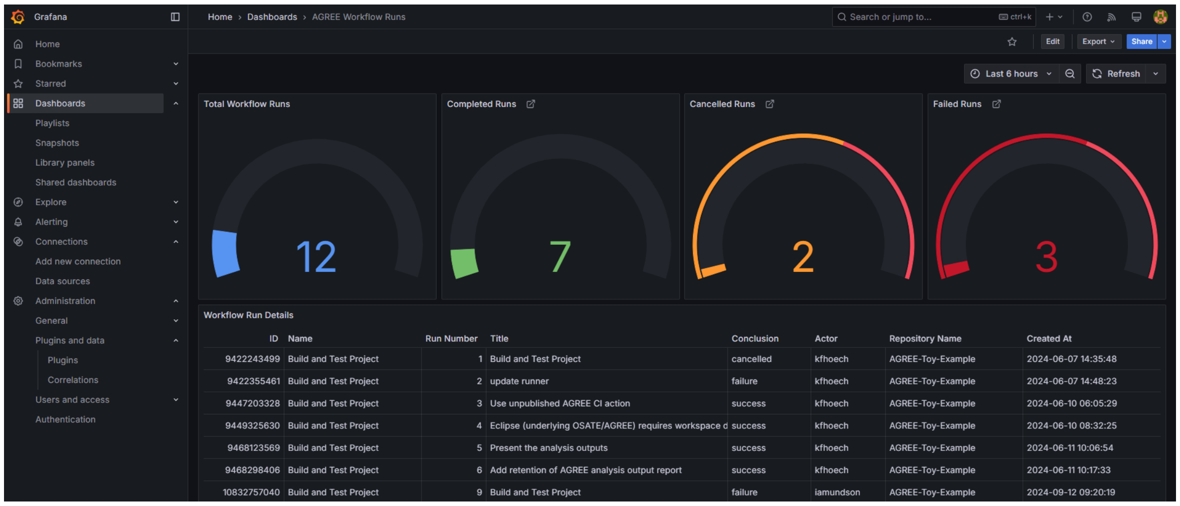
```
fn timeTriggered<API: Manage_Heat_Source_i_Full_Api>(&mut self,
                                                     api: &mut Manage_Heat_Source_i_Application_Api<API>)
  requires
     old(api).lower_desired_temp.degrees <= old(api).upper_desired_temp.degrees
  ensures
    // guarantee lastCmd
    // Set lastCmd to value of output Cmd port
    (self.lastCmd == api.heat_control)
  && // case REQ_MHS_1
     // If the Regulator Mode is INIT, the Heat Control shall be set to Off.
     ((api.regulator_mode == data::Regulator_Mode::Init_Regulator_Mode) ==>
      (api.heat_control == data::OnOff::Off))
  && // case REQ_MHS_2
     // If the Regulator Mode is NORMAL and the Current Temperature is less than
     // the Lower Desired Temperature, the Heat Control shall be set to On.
     ((api.regulator_mode == data::Regulator_Mode::Normal_Regulator_Mode &&
       api.current_tempWstatus.degrees < api.lower_desired_temp.degrees) ==>
      (api.heat_control == data::OnOff::Onn))
  && [...]
{
  // -------------- Get values of input ports ------------------
  let lower: data::Temp_i = api.get_lower_desired_temp(); // gives lower <= api.upper_desired_temp.degrees
  let upper: data::Temp_i = api.get_upper_desired_temp(); // gives api.lower_desired_temp.degrees <= upper

  let regulator_mode: data::Regulator_Mode = api.get_regulator_mode();
  let currentTemp: data::TempWstatus_i = api.get_current_tempWstatus();

  //================ compute / control logic ==========================

  // current command defaults to value of last command (REQ-MHS-4)
  let mut currentCmd: data::OnOff = self.lastCmd;

  match regulator_mode {
    // ----- INIT Mode --------
    data::Regulator_Mode::Init_Regulator_Mode => {
      // REQ-MHS-1
      currentCmd = data::OnOff::Off;
    },
    // ------ NORMAL Mode -------
    data::Regulator_Mode::Normal_Regulator_Mode => {
      if (currentTemp.degrees < lower.degrees) {
        assert(api.current_tempWstatus.degrees < api.lower_desired_temp.degrees);
        // REQ-MHS-2
        currentCmd = data::OnOff::Onn;
      } [...]
    } [...]
  }
}
```

Fig. 4. Rust code with Verus verification annotations automatically generated from the SysML v2 model of Fig. 1.

ming language, and demonstrated its property specification and proof capability, all in the context of a Continuous Integration/Continuous Deployment (CI/CD) framework of the type used by avionics developers. Our framework provides allocation to/traceability from the corresponding software and hardware design, particularly ensuring that the system model and design/implementation remain "in-sync" during product development. Additionally, our toolchain provides guarantees that assurance properties established at the system architecture model level hold at the design/implementation level. Our method applies formal verification at each level in the design to assure the design at that level before proceeding to the next level. The capability of our tools to automatically generate verification-enhanced memory-safe language code from architecture models featuring assume/guarantee contracts is unique in our experience.

In future work, we will continue the work to integrate our contract language and assurance case annotations with SysML v2. We will complete the harmonization of the AGREE and GUMBO contract languages, as well as make the translation from System Architecture model contracts to Verus Rust contracts more automatic. We will complete the standardization work to ensure that AADL semantics can be applied to SysML v2 models. We will expand our environment to embrace behavioral aspects of SysML v2 not found in AADL, such as state machines. Additionally, we will be transitioning the automated property-based testing infrastructure previously developed for AADL/HAMR/Slang [34] in order to automatically synthesize property-based tests for SysML v2 and Rust.

On the CI/CD integration front, we will demonstrate the



Total Workflow Runs 12

Completed Runs 7

Cancelled Runs 2

Failed Runs 3

Workflow Run Details

ID	Name	Run Number	Title	Conclusion	Actor	Repository Name	Created At
9422243499	Build and Test Project	1	Build and Test Project	cancelled	kfhoech	AGREE-Toy-Example	2024-06-07 14:35:48
9422355461	Build and Test Project	2	update runner	failure	kfhoech	AGREE-Toy-Example	2024-06-07 14:48:23
9447203328	Build and Test Project	3	Use unpublished AGREE CI action	success	kfhoech	AGREE-Toy-Example	2024-06-10 06:05:29
9449325630	Build and Test Project	4	Eclipse (underlying OSATE/AGREE) requires workspace d	success	kfhoech	AGREE-Toy-Example	2024-06-10 08:32:25
9468123569	Build and Test Project	5	Present the analysis outputs	success	kfhoech	AGREE-Toy-Example	2024-06-11 10:06:54
9468298406	Build and Test Project	6	Add retention of AGREE analysis output report	success	kfhoech	AGREE-Toy-Example	2024-06-11 10:17:33
10832757040	Build and Test Project	9	Build and Test Project	failure	iamundson	AGREE-Toy-Example	2024-09-12 09:20:19

Fig. 5. Continuous Integration/Continuous Deployment Dashboard for ProofOps.

integration of our formal verification tools with a production Collins Aerospace CI/CD environment, including a "ProofOps Dashboard" that provides a real-time summary of the state of the various proof activities. Finally, we will demonstrate the use of our high-assurance system architecture modeling, detailed design, and implementation toolchain on Collins Aerospace safety- and securty-critical avionics applications currently in development. This includes the use of SysML v2 modeling, assume/guarantee contracts, and HAMR skeletal code generation to Rust/Verus, all hosted on the seL4 microkernel, and running on production Collins Aerospace hardware. Importantly, the majority of the development will be performed by product development engineers, with minimal assistance by our formal methods researchers.

IX. Acknowledgments

This work was funded by DARPA contract FA8750-24-9-1000. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We wish to particularly acknowledge Brad Martin for his vision and hard work to establish the PROVERS program during his time at DARPA, as well as his leadership role in advancing the state of the art in formal methods throughout his years of U.S. Government service. Thanks also go to the anonymous reviewers for their helpful comments.

References

[1] *PROVERS: Pipelined Reasoning of Verifiers Enabling Robust Systems*, Defense Advanced Research Projects Agency, March 2023. [Online]. Available: https://www.darpa.mil/research/programs/pipelined-reasoning-of-verifiers-enabling-robust-systems

[2] D. Cofer, I. Amundson, J. Babar, D. Hardin, K. Slind, P. Alexander, J. Hatcliff, Robby, G. Klein, C. Lewis, E. Mercer, and J. Shackleton, "Cyber assured systems engineering at scale," in *IEEE Security & Privacy*, May/June 2022, pp. 52–64.

[3] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[4] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, 2010. [Online]. Available: http://doi.acm.org/10.1145/1743546.1743574

[5] *OMG Systems Modeling Language (SysML), Version 2.0 Beta 2*, Object Management Group, February 2024. [Online]. Available: https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF

[6] J. Hugues, "Aadlv2 library for sysmlv2," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2023-TN-001, April 2023, approved for public release and unlimited distribution. [Online]. Available: https://apps.dtic.mil/sti/trecms/pdf/AD1207053.pdf

[7] S. Hallerstede and J. Hatcliff, "A mechanized semantics for component-based systems in the hamr aadl runtime," *Science of Computer Programming*, vol. 245, p. 103312, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0167642325000516

[8] D. D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart, "A formal approach to constructing secure air vehicle software," *Computer*, vol. 51, no. 11, pp. 14–23, 2018.

[9] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM*, 1987.

[11] J. Hatcliff, D. Stewart, J. Belt, Robby, and A. Schwerdfeger, "An AADL contract language supporting integrated model- and code-level verification," *Ada Letters*, vol. 42, no. 2, p. 45–54, April 2023. [Online]. Available: https://doi.org/10.1145/3591335.3591339

[12] J. Hatcliff, J. Belt, Robby, and T. Carpenter, "HAMR: An AADL multi-platform code generation toolset," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 274–295.

[13] Robby and J. Hatcliff, "Slang: The Sireum programming language," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 253–273.

[14] I. Velickovic, *Microkit User Manual*, 2025. [Online]. Available: https://github.com/seL4/microkit/blob/main/docs/manual.md

[15] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing

Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190539

[16] A. Petz and P. Alexander, "An infrastructure for faithful execution of remote attestation protocols," in *Proceedings of the 13th NASA Formal Methods Symposium (NFM 2021)*, May 2021.

[17] *Software Memory Safety*, National Security Agency, November 2022. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[18] *Back to the Building Blocks: A Path Toward Secure and Measurable Software*, White House Office of the National Cyber Director, February 2024. [Online]. Available: https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[19] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192.

[20] J. V. Stoep and S. Hines, "Rust in the Android platform," April 2021. [Online]. Available: https://security.googleblog.com/2021/04/rust-in-android-platform.html

[21] S. Miller and C. Lerche, "Sustainability with Rust," February 2022. [Online]. Available: https://aws.amazon.com/blogs/opensource/sustainability-with-rust/

[22] R. Amadeo, "Google is now writing low-level Android code in Rust," April 2021. [Online]. Available: https://arstechnica.com/gadgets/2021/04/google-is-now-writing-low-level-android-code-in-rust/

[23] M. Miller, "A proactive approach to more secure code," July 2019. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

[24] T. Claburn, "Microsoft is busy rewriting core Windows code in memory-safe Rust," April 2023. [Online]. Available: https://www.theregister.com/2023/04/27/microsoft_windows_rust/

[25] D. S. Hardin, "Verified hardware/software co-assurance: Enhancing safety and security for critical systems," in *Proceedings of the 2020 IEEE Systems Conference*, 2020.

[26] ——, "Hardware/software co-assurance using the Rust programming language and ACL2," in *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-22)*, May 2022, pp. 202–216.

[27] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, April 2023.

[28] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman, "SPARK 2014 and GNATprove," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, 2015.

[29] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Proceedings of the 22nd European Symposium on Programming*, ser. LNCS, M. Felleisen and P. Gardner, Eds., vol. 7792. Springer, March 2013, pp. 125–128.

[30] X. Denis, *Creusot*, September 2022. [Online]. Available: https://github.com/xldenis/creusot

[31] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The Prusti project: Formal verification for Rust (invited)," in *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108.

[32] Y. Matsushita, T. Tsukada, and N. Kobayashi, "Rusthorn: CHC-based verification for Rust programs," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 4, oct 2021.

[33] *Announcing the Kani Rust Verifier Project*, Amazon Web Services, May 2022. [Online]. Available: https://model-checking.github.io/kani-verifier-blog/2022/05/04/announcing-the-kani-rust-verifier-project.html?fbclid=IwAR2M_B1IEBfkVhIXSuuAxt3McC_QpUnTuzDq9jG40HOaJzxw8z1Nw9XU_i4

[34] J. Hatcliff, J. Belt, Robby, and D. Hardin, "Integrated contract-based unit and system testing for component-based systems," in *Proceedings of the 16th NASA Formal Methods Symposium (NFM 2024)*, N. Benz, D. Gopinath, and N. Shi, Eds., June 2024.