Resolute Assurance Arguments for Cyber Assured Systems Engineering

Isaac Amundson and Darren Cofer
{isaac.amundson,darren.cofer}@collins.com
Collins Aerospace
Minneapolis, Minnesota, USA

ABSTRACT

Resolute is a tool and language for embedding an assurance argument in a system architecture model and evaluating the validity of the associated evidence. In this paper we report on a number of extensions to Resolute that support systems engineers in developing safe and cyber-resilient systems. System requirements are imported as assurance goals to be satisfied. Architectural transforms are applied to the system model to address these requirements, while corresponding assurance strategies and evidence are automatically added to document how the requirements have been satisfied. Subsequent changes to the model that invalidate any of the assurance claims can be detected and corrected. We also use Resolute to check that the model satisfies rules for code generation and other modeling guidelines. We conclude with an application of the Resolute assurance process to the design of a mission planning system for an unmanned air vehicle.

KEYWORDS

assurance case, cyber-security, formal methods

ACM Reference Format:
Isaac Amundson and Darren Cofer. 2021. Resolute Assurance Arguments for Cyber Assured Systems Engineering. In *Proceedings of DESTION 2021: Design Automation for CPS and IoT (DESTION 2021).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, aerospace stakeholders have become aware that avionics systems are subject to possible cyber-attack just like other cyber-physical systems. In addition to being fault-tolerant, safety-critical avionics systems must also be be *cyber-resilient*. Cyber-resiliency means that the system is tolerant to cyberattacks just as safety-critical systems are tolerant to random faults: they recover and continue to execute their mission function, or safely shut down, as requirements dictate.

Unfortunately, systems engineers are currently given few development tools to help answer even basic questions about potential vulnerabilities and mitigations, and instead rely on process-oriented checklists and guidelines. Cyber vulnerabilities are often discovered during penetration testing late in the development process. Worse yet, they may be discovered after the product has been fielded, necessitating extremely expensive and time-consuming remediation. This is not a sustainable development model.

In the DARPA Cyber Assured Systems Engineering (CASE) program, our team is developing design, analysis, and verification tools that enable systems engineers to *design-in* cyber-resiliency for complex cyber-physical systems. We have produced a prototype Model-Based Systems Engineering (MBSE) environment called *BriefCASE* which is based on the Architecture Analysis and Design Language (AADL) [14]. BriefCASE extends the Open Source AADL Tool Environmnet (OSATE) to add new design, analysis, and code generation capabilities targeted at building cyber-resilient systems.

BriefCASE provides access to two analysis tools (GearCASE [13] and DCRYPPS [12]) that can examine AADL models to detect potential cyber vulnerabilities and suggest requirements for mitigation. A library of architectural transforms guides systems engineers through automated model transformations that modify the architecture to address these requirements, possibly inserting new high-assurance components into the system. Implementations for these new high-assurance components are synthesized from formal specifications using the Semantic Properties for Language and Automata Theory (SPLAT) tool [16]. Formal verification that the transformed system model satisfies its cyber requirements is accomplished via model checking using the Assume Guarantee Reasoning Environment (AGREE) [18]. Cyber-resilient code implementing the verified model is automatically generated using the High Assurance Modeling and Rapid Engineering for Embedded Systems (HAMR) toolkit [7]. If desired, this code can be targeted to the formally verified seL4 secure microkernel [11].

A novel aspect of our approach is the use of an assurance argument embedded in the architecture model itself to capture and document the design decisions made during this process, along with associated rationale.

We developed the *Resolute* language and tool [5] as a way to help developers create an assurance argument describing the steps taken during the design process to make the system safe and secure. Rather than being a separate document, a Resolute assurance case is part of the architecture model and can refer to elements within the model. Since it is not a static representation, it can ensure that the assurance argument remains consistent with the evolving design.

Assurance cases have a large and well-developed literature. Patterns for assurance case argumentation have been considered in [3, 8, 9, 17]. An approach to apply and evolve assurance cases as part of system design is found in [6], which is similar to the process we have used in applying Resolute. Additional related tools and studies are described in [5].

DESTION 2021, May 18, 2021, Nashville, TN

ACM ISBN 978-1-4503-8316-5…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RESOLUTE

An important aspect of our work on CASE has been to structure formalizations and proofs by following the AADL description of the system. In other work, we did this through the use of formal assume-guarantee contracts that correspond to the requirements for each component [2]. We have found that in assuring the cybersecurity properties of aircraft designs we need to integrate different kinds of evidence with varying levels of formality. This has been our motivation to explore assurance case methods.

Resolute is an assurance case language and tool which is based on AADL architectural models. In developing Resolute, we have followed the same approach of embedding the proof in the architectural model, tightly coupling terms in the assurance case with evidence derived directly from the system design artifacts. This ensures that we maintain consistency between the system design and its associated assurance case(s). Design changes that might invalidate some aspect of an assurance case can be immediately flagged by our tool for correction.

Users formulate claims and rules for satisfying those claims, which Resolute uses to construct assurance cases. Both the claims and rules are parameterized by variables which are instantiated using elements from the models. This connects the assurance case directly to the AADL model and means that changes to the AADL model can result in changes to the assurance case. In Resolute, each claim corresponds to a first-order predicate. Logically, these rules correspond to global assumptions that have the form of an implication with the predicate of interest as the conclusion. This means that a small reusable set of rules can result in a large assurance case since each rule may be applied multiple times to different parts of the architecture model.

Resolute allows users to incorporate computations in their assurance arguments. Usually these computations are based on querying the model structure. Analyses performed by external tools can also be incorporated in Resolute as computations. This is useful for incorporating as evidence that results from existing analysis tools for checking properties such as schedulability or resource allocation.

Resolute syntax has been extended to support construction of assurance cases that comply with the Goal Structuring Notation (GSN) v2 standard [15]. Claims can now be expressed as *goals* and *strategies*, and they can contain attributes such as *context*, *assumptions*, and *justification*. Claims can be marked *undeveloped*, which Resolute interprets as an unsupported claim, or with a *solution*, which is an explicit assertion that the claim is supported.

Support for GSN enables Resolute results to be exported to the AdvoCATE tool [4]. A new export option has been included, which generates an *argument* file that can be imported directly into an AdvoCATE project and merged with an existing assurance case.

The Resolute package also includes a new linter tool for AADL models, called *Resolint*. Resolint provides a language for specifying rules that correspond to modeling guidelines, as well as a checker for evaluating whether a model complies with the rules. Results of the Resolint analysis are displayed to the user, and can even be incorporated as evidence in a Resolute assurance argument. Rule violations indicate severity, and are linked to the model element that is out of compliance with the rule.

3 BRIEFCASE OVERVIEW

Our BriefCASE toolchain provides systems engineers with a workflow and tool support for developing products with inherent cyber-resiliency. BriefCASE is predicated on an MBSE process, in which models are the primary vehicle for communication and understanding among the parties tasked with designing the system. Furthermore, MBSE models are the primary design artifacts used for analysis, verification, testing, and code generation.

The BriefCASE workflow starts with the development of an AADL model of the system architecture. BriefCASE is implemented as a set of plugins that work with OSATE, the flagship tool for AADL modeling. Once an architecture model has been created, it can be analyzed in various ways (e.g., resource usage, information flow, latency) to determine whether the initial design is acceptable.

BriefCASE integrates tools that analyze the architecture model for cybersecurity vulnerabilities and generate a set of requirements that, when addressed, will mitigate those vulnerabilities. The generated requirements are imported into the model and represented as goals in a Resolute assurance case. As a requirement is addressed in the design, the assurance case is updated with evidence, either taken directly from the model or supporting development process outputs, necessary to support the claim. In this manner, the assurance case is co-developed alongside the system design, and can be automatically evaluated throughout development.

BriefCASE provides systems engineers with a requirements management interface (Figure 1) for viewing the generated requirements and importing them into the model so they can be addressed. The interface enables engineers to select the requirements they wish to import and assign them unique IDs, or omit them with rationale. A document of the omitted requirements and rationale is maintained, and may be a required development artifact for some certification domains. Some requirements can also be formalized as assume-guarantee contracts, enabling formal analysis in AGREE. Such a requirement will be imported with an associated formal AGREE contract.

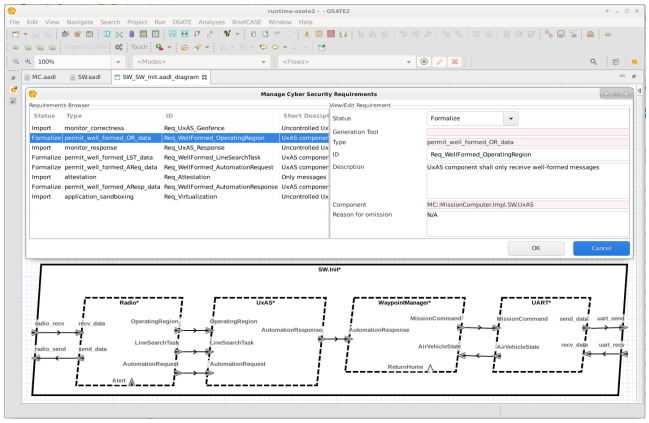

Figure 1: Requirements management interface.

A BriefCASE project contains a repository for requirements. Imported requirements are represented as Resolute goals to be satisfied. For example, the well-formedness requirement selected

in Figure 1 is imported as the Resolute goal shown in Figure 2. Initially, the goal is marked *undeveloped*, and does not contain any evidential statements for Resolute to evaluate in order to determine whether the goal has been satisfied. Running Resolute at this time will therefore produce a failed assurance case.

```
goal Req_WellFormed_OperatingRegion(comp_context : component) <=
    ** "UxAS component shall only receive well-formed messages" **
    context Generated_On : "January 29, 2021";
    context Req_Component : "MC::MissionComputer.Impl.SW.UxAS";
    undeveloped
```

Figure 2: Resolute well-formedness requirement.

To address the new requirement, the architecture will need to be transformed in such a way as to harden the design against the vulnerability. BriefCASE provides a library of model transformations for addressing common cyber vulnerabilities. The transformations are automated by the tool, resulting in a hardened model that is correct-by-construction. For example, ensuring a component only receives well-formed messages can be accomplished by the insertion of a high-assurance filter. The BriefCASE Filter transform wizard (Figure 3) enables the configuration of filter component properties, including the filter behavioral specification, which is represented in the AGREE language.

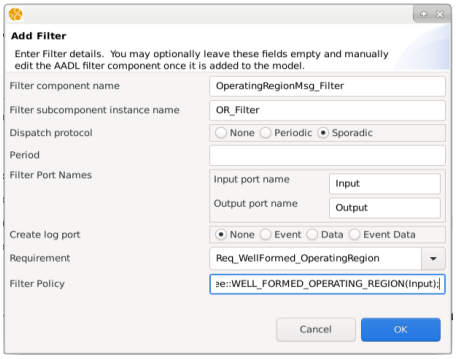

Figure 3: Filter transform wizard.

BriefCASE inserts a new filter component into the model, sets the component properties, and establishes the appropriate connections to source and destination components. The filter specification is inserted into an AGREE annex, enabling both formal analysis of the model as well as providing the behavioral specification for a provably correct synthesis of the component implementation via the SPLAT plugin.

The transformation also updates the Resolute goal with new evidential statements pointing to evidence that the model has indeed been hardened against the vulnerability and the requirement has been satisfied (as shown in Figure 4). For example, the `add_filter` strategy is included in a library of built-in Resolute transform rules and provides Resolute with the logical instructions for evaluating if the top-level goal has been satisfied. The `add_filter` definition (shown in Figure 4) includes the following sub-goals:

- `filter_exists` - the filter component exists in the model
- `filter_not_bypassed` - there is no alternate pathway in the model that can bypass the filter
- `filter_implemented_correctly` - the filter has been implemented correctly

```
goal Req_WellFormed_OperatingRegion(comp_context : component, filter : component,
                                    conn : connection, message_type : aadl) <=
    ** "UxAS component shall only receive well-formed messages" **
    context Generated_On : "January 29, 2021";
    context Req_Component : "MC::MissionComputer.Impl.SW.UxAS";
    add_filter(comp_context, filter, conn, message_type)


-- Strategy for proper insertion of a filter
strategy add_filter(comp_context : component, filter : component,
                    conn : connection, msg_type : aadl) <=
    ** "Filter is properly inserted" **
    filter_exists(filter, comp_context, conn) and
    component_not_bypassed(filter, comp_context, msg_type) and
    component_implemented(filter)
```

Figure 4: Updated well-formedness claim.

The first two sub-goals are supported by evidence obtained by examining the structure of the model, while the last is determined by examining the output of the synthesis tool. This approach follows the model-based decomposition pattern based on [1], and is representative of all BriefCASE transform assurance strategies. If at a later time during development the model is inadvertently altered in a way that renders the transformation ineffective, Resolute will be unable to substantiate the evidential statements, and therefore produce a failing assurance case.

The third subgoal is satisfied by SPLAT. SPLAT not only generates the implementation code for high-assurance components such as filters, monitors, and gates, but it also produces a proof that this code correctly implements its AGREE specification. Resolute uses the existence of the SPLAT proof as evidence that the component was implemented correctly.

After all imported requirements have been addressed, no new requirements are generated from subsequent analyses, the model passes formal verification, and compliance with modeling guidelines has been ascertained, then the system can be built and deployed. HAMR is a code generation and system build framework included in BriefCASE. HAMR supports development of new components and wrapping of legacy components by generating code that provides the interfacing infrastructure between components. It translates the AADL system model to code that implements the threading infrastructure and inter-component communication that is consistent with the AADL computational model. It also generates a correspondence proof showing that the generated code preserves the information flow specified in the AADL model. Resolute can check for the existence of this proof as evidence that the build has been carried out correctly.

After the build has completed, Resolute can be run one last time to generate the full assurance argumentation associated with the BriefCASE workflow. The resulting assurance case sub-tree can then be integrated into a full system assurance case for evaluation.

4 APPLICATION

In this section, we demonstrate the role of Resolute in the BriefCASE toolchain by designing a UAV surveillance application in which a UAV receives commands from a ground station to conduct

surveillance along a geographical feature such as a river. The onboard mission computer then generates a flight plan consisting of a series of waypoints that the UAV must traverse to complete its mission. The UAV is also given a set of *keep-in* and *keep-out* zones that may constrain its flight path.

We have modeled the system architecture of the UAV in AADL. It includes a mission computer for communicating with the ground station and generating flight plans, and a flight control computer for UAV navigation. The mission computer architecture model includes hardware components such as a processor, memory, and communication devices, as well as software. The initial software architecture model (shown in Figure 1) contains drivers for communication with the ground station and flight control computer, a Waypoint Manager component that provides flight plan coordinates to the flight control computer, and the flight planner. The flight planner is the open-source UxAS software developed by AFRL [10].

In BriefCASE, we analyze the model using one (or more) of the integrated cybersecurity analysis tools, which generates a list of new requirements. To satisfy these requirements we will need to mitigate the vulnerabilities discovered by the analysis by modifying the design. In total we import eight new cyber requirements, including one *sandboxing* requirement, four *well-formedness* requirements, two requirements for *monitoring* the behavior of the open-source UxAS component, and an *attestation* requirement for ensuring the ground station software has not been tampered with. As described in Section 3, requirements are imported into the model as goals in a Resolute assurance case. Because we can run Resolute at any time during development, we can easily determine for a given snapshot of the model which requirements are not yet supported by evidence. For example, running Resolute on the snapshot immediately after importing the new requirements will produce the results shown in Figure 5.

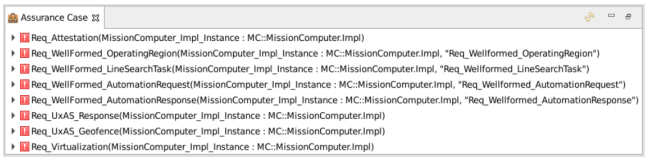

Figure 5: The initial Resolute evaluation fails.

Resolute produces a failing assurance case because it does not have enough information to evaluate whether the assurance goals representing the new requirements have been satisfied. BriefCASE provides a library of model transformations that mitigate several classes of cyber vulnerabilities. Not only does BriefCASE transform the model by modifying the architecture in a manner that addresses the vulnerability, but it also modifies the Resolute assurance argument by adding new strategies describing the evidence that is required to support the goal. This is possible because the transform wizard links the mitigation to the Resolute requirement that is driving it.

The *sandboxing* requirement was generated because UxAS is untrusted code. We did not develop it ourselves and it is open source software that could potentially contain malicious code. To mitigate this vulnerability, we place it in a virtual machine so that the effects of any such malicious code will be contained. The Virtualization transform adds a virtual processor to the architecture and updates processor bindings. The Resolute requirement is then updated to check that the virtual machine is bound to the specified processor, and that the target components are bound to the virtual processor (and do not execute on any other processor). After applying the Virtualization transform, the Resolute assurance argument for the sandboxing requirement passes, as shown in Figure 6.

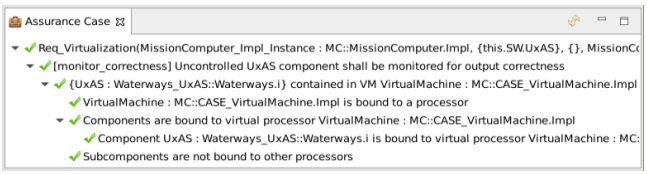

Figure 6: Assurance argument after applying the Virtualization transform.

Although we have now placed UxAS in a virtual machine, it is still communicating with other critical components in our system. The intent of the well-formedness requirement is to prevent malformed messages that could be sent from an infected UxAS from reaching the Waypoint Manager and causing a buffer overrun or code injection attack. By placing a filter on the connection joining these two components, such an attack could be mitigated. The three other well-formedness requirements that were generated apply to the incoming UxAS connections.

The Filter transform (described in Section 3) is applied for each requirement, inserting filter components on the incoming and outgoing UxAS connections. The filter behavior for each component is specified in AGREE. Not only does this enable formal verification within the modeling environment, but it also provides a means for synthesizing the component implementation in a provably correct manner using the SPLAT tool. Because SPLAT is integrated with BriefCASE, the proof it emits when synthesizing component code is used as evidence in the Resolute goal for the corresponding mitigation. When Resolute evaluates whether such a goal is supported by evidence, it checks for the existence of the synthesis proof in addition to verifying the architecture is correct. After performing all four Filter transforms, the resulting assurance arguments are shown in Figure 7 (some branches are truncated for clarity).

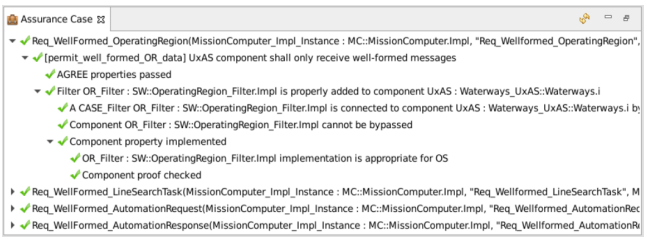

Figure 7: Assurance arguments after applying the Filter transforms.

In addition to monitoring the UxAS output for malformed messages, we must also monitor for suspicious behavior. This requires

adding components for detecting that UxAS has crashed, as well as monitoring the correctness of the flight plans it produces. The Monitor transform is applied for this class of mitigation. We perform two transforms. The first adds a *response* monitor to send an alert if UxAS does not emit a response within a set amount of time from receiving a request. The second adds a *geofence* monitor to ensure that generated flight plans are compliant with the specified keep-in and keep-out zones. Similar to the filters, the monitor behavior is specified in AGREE, enabling formal verification and provably correct synthesis. The associated Resolute requirement is similarly updated to check the correctness of the architecture, that the AGREE analysis passes, and for implementation correctness. The monitor assurance arguments appear as shown in Figure 8.

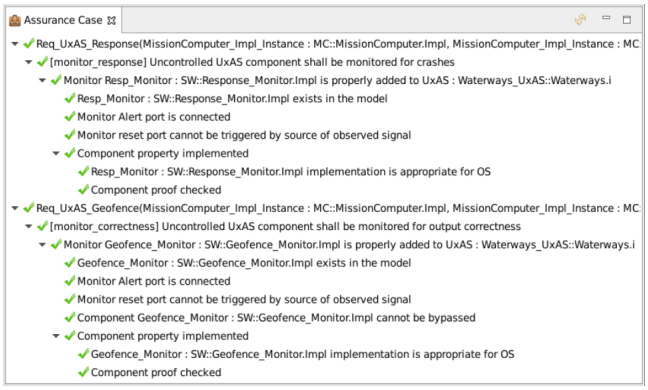

Figure 8: Assurance arguments after applying the Monitor transforms.

The requirements that have been addressed so far mitigate vulnerabilities related to malformed messages and malicious behavior *on board* the UAV. But we also want to protect against a compromised ground station that could potentially transmit well-formed, but malicious commands. The final cyber-requirement is mitigated by the Attestation transform, which adds two components to the UAV software: an Attestation Manager for evaluating remote systems like the ground station, and an Attestation Gate for blocking messages from sources that have not been white-listed by the Attestation Manager. With the attestation requirement addressed, the corresponding assurance argument is shown in Figure 9.

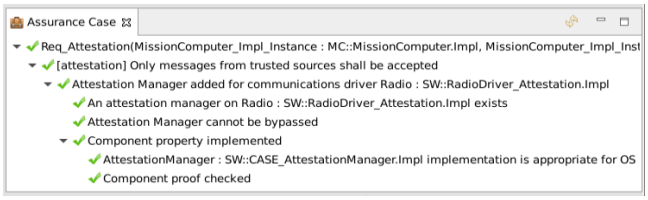

Figure 9: Assurance argument after applying the Attestation transform.

After transforming the model to address its cyber requirements, the software architecture now appears as shown in Figure 10. The components in green were added to the model by way of an automated BriefCASE transform and are critical for mitigating cyber attacks. We formally verify the model with AGREE to show that all of the contracts are satisfied, and Resolute automatically evaluates the results in the assurance case. Because it is imperative that the high-assurance component implementations are also correct, we run the SPLAT tool to produce provably-correct code. The synthesized code is output to a directory in the build file system with the location specified for each component in the model. The corresponding correctness proof is used in our assurance case as additional evidence that the vulnerability has been properly mitigated.

Most certification standards, as well as industry best practices, recommend that modeling activities comply with a set of modeling guidelines. Running Resolint on our UAV model confirms that we are in full compliance with our modeling guidelines and HAMR code generation preconditions. The Resolint results are yet another piece of assurance evidence automatically inserted into the assurance case and evaluated by Resolute.

Once we have determined that the model is correct and satisfies its cyber requirements, we run HAMR to generate the component stubs and infrastructure code necessary to enable component communication and execution according to a specified schedule.

Running Resolute one last time and exporting to the AdvoCATE tool produces the assurance argument shown in Figure 11. Blue elements correspond to model correctness and orange elements correspond to implementation correctness. Due to space limitations, a comprehensive argument cannot be displayed. All the goals are supported by evidence generated by the BriefCASE tools, which provides us with confidence that the deployed system will be resilient to cyber attacks that could otherwise lead to mission failure.

Videos demonstrating the use of the BriefCASE tools to build this UAV example are available at loonwerks.com/projects/case.

5 CONCLUSION

BriefCASE can help systems engineers build complex cyberphysical systems that are cyber-resilient by design. Cyber requirements are captured as assurance goals to be met by the system design. The architectural transforms and code synthesis plugins automatically create a corresponding assurance argument in the background, gathering the associated evidence from the model structure and different analysis and proof tools.

Next steps include further development and capture of evidence related to the build process. This includes connecting the correspondence proof generated by HAMR to the separation guarantees provided by the seL4 secure kernel, demonstrating that the information flow properties described in the AADL model are preserved in the executable code. In the final phase of the CASE program we will also be applying the BriefCASE tools to new connectivity features being added to the avionics system of a military helicopter.

ACKNOWLEDGMENTS

This work was funded by DARPA contract HR00111890001. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

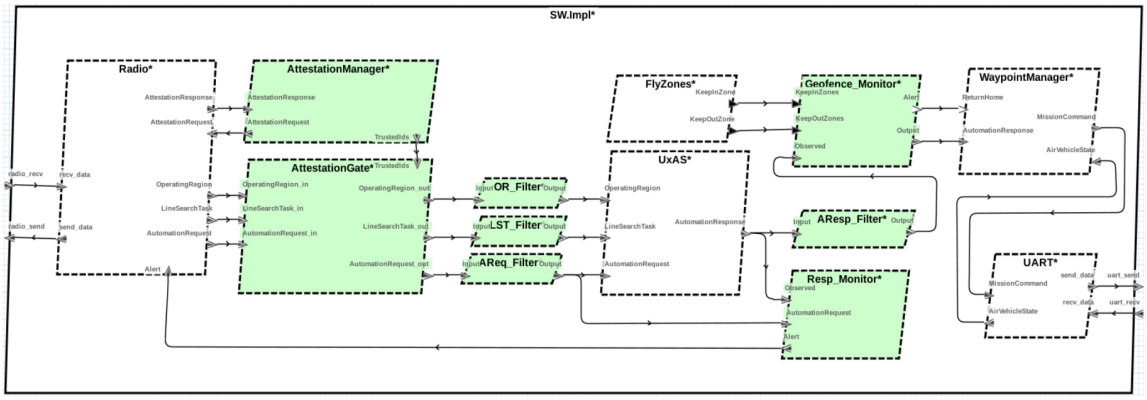

Figure 10: Cyber-resilient software architecture.

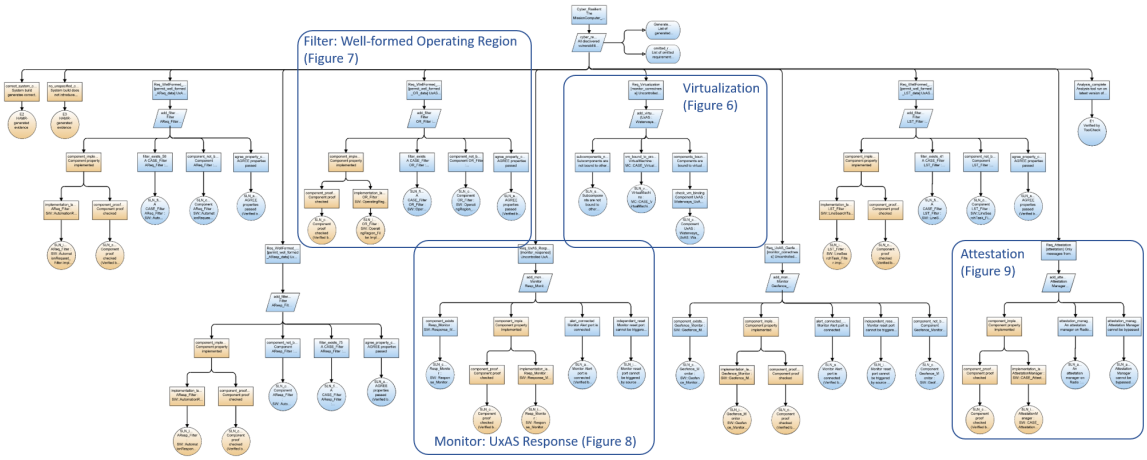

Figure 11: Cyber-resiliency assurance argument for the hardened system.

REFERENCES

[1] Anaheed Ayoub, BaekGyu Kim, Insup Lee, and Oleg Sokolsky. 2012. A Safety Case Pattern for Model-Based Development Approach. 141–146.

[2] Darren D. Cofer, Andrew Gacek, John Backes, Michael W. Whalen, Lee Pike, Adam Foltzer, Michal Podhradsky, Gerwin Klein, Ihor Kuz, June Andronick, Gernot Heiser, and Douglas Stuart. 2018. A Formal Approach to Constructing Secure Air Vehicle Software. *Computer* 51, 11 (2018), 14–23.

[3] E. Denney and G. Pai. 2013. A Formal Basis for Safety Case Patterns. In *Proceedings of the 2013 International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (Toulouse, France).

[4] Ewen Denney and Ganesh Pai. 2018. Tool Support for Assurance Case Development. *Automated Software Engineering* 25 (09 2018).

[5] Andrew Gacek, John Backes, Darren D. Cofer, Konrad Slind, and Mike Whalen. 2014. Resolute: an assurance case language for architecture models. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 19–28.

[6] P. Graydon, J. Knight, and E. Strunk. 2007. Assurance Based Development of Critical Systems. In *2007 International Symposium on Dependable Systems and Networks (DSN)* (Edinburgh, Scotland).

[7] HAMR 2021. *High Assurance Modeling and Rapid engineering for embedded systems.* Retrieved Feb 26, 2021 from http://sireum.hamr.org

[8] R. Hawkins, K. Clegg, R. Alexander, and T. Kelly. 2011. Using a Software Safety Argument Pattern Catalogue: Two Case Studies. In *Proceedings of the 2011 International Conference on Computer Safety, Reliability and Security (SAFECOMP)*.

[9] T. Kelly and J. McDermid. 1997. Safety case construction and reuse using patterns. In *Proceedings of the 1997 International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*.

[10] Derek B. Kingston, Steven Rasmussen, and Laura R. Humphrey. 2016. Automated UAV tasks for search and surveillance. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*. IEEE, 1–8.

[11] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220.

[12] Robert Laddaga, Paul Robertson, Howard E. Shrobe, Dan Cerys, Prakash Manghwani, and Patrik Meijer. 2019. Deriving Cyber-security Requirements for Cyber Physical Systems. *CoRR* abs/1901.01867 (2019). arXiv:1901.01867 http://arxiv.org/abs/1901.01867

[13] Mitchell D. Patten, T. and C. Call. 2020. Cyber Attack Grammars for Risk-Cost Analysis. In *Proceedings of the 15th International Conference on Cyber Warfare and Security*. Norfolk, VA.

[14] SAE. 2009. *Architecture Analysis and Design Language (AADL).* Technical Report AS-5506. SAE International. https://www.sae.org/standards/content/as5506a/

[15] SCSC-141B. 2011. *Goal Structuring Notation Community Standard (Version 2).* The Assurance Case Working Group.

[16] Konrad Slind. 2020. Take a Seat: Security-Enhancing Architecture Transforms. In *Proceedings of the 20th High Confidence Software and Systems Conference*. https://cps-vo.org/hcss2020/slind

[17] L. Sun, O. Lisagor, and T. Kelly. 2011. Justifying the Validity of Safety Assessment Models with Safety Case Patterns. In *Proceedings of the 6th IET System Safety Conference* (Birmingham, UK).

[18] Michael W. Whalen, Andrew Gacek, Darren D. Cofer, Anitha Murugesan, Mats Per Erik Heimdahl, and Sanjai Rayadurgam. 2013. Your "What" Is My "How": Iteration and Hierarchy in System Design. *IEEE Softw.* 30, 2 (2013), 54–60.