systems

MDPI

Article

VERDICT: A Language and Framework for Engineering Cyber Resilient and Safe System †

Baoluo Meng [1,*], Daniel Larraz [2], Kit Siu [1], Abha Moitra [1], John Interrante [1], William Smith [1], Saswata Paul [1], Daniel Prince [3], Heber Herencia-Zapana [1], M. Fareed Arif [2], Moosa Yahyazadeh [2], Vidhya Tekken Valapil [1], Michael Durling [1], Cesare Tinelli [2] and Omar Chowdhury [2]

[1] GE Research, 1 Research Cir, Niskayuna, NY 12309, USA; siu@ge.com (K.S.); moitraa@ge.com (A.M.); interran@research.ge.com (J.I.); william.D.smith@ge.com (W.S.); saswata.paul@ge.com (S.P.); heber.herencia-zapana@ge.com (H.H.-Z.); vidhya.valapil@ge.com (V.T.V.); durling@ge.com (M.D.)
[2] Department of Computer Science, The University of Iowa, Iowa City, IA 52242, USA; daniel-larraz@uiowa.edu (D.L.); muhammad-arif@uiowa.edu (M.F.A.); moosa-yahyazadeh@uiowa.edu (M.Y.); cesare-tinelli@uiowa.edu (C.T.); omar-chowdhury@uiowa.edu (O.C.)
[3] GE Aviation, 3290 Patterson Ave SE, Grand Rapids, MI 49512, USA; Daniel.Prince@ge.com
* Correspondence: baoluo.meng@ge.com
† This manuscript is an extension version of the conference paper: K. Siu et al. "Architectural and Behavioral Analysis for Cyber Security." 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), San Diego, CA, USA, 2019.

Abstract: The ever-increasing complexity of cyber-physical systems is driving the need for assurance of critical infrastructure and embedded systems. However, traditional methods to secure cyber-physical systems—e.g., using cyber best practices, adapting mechanisms from information technology systems, and penetration testing followed by patching—are becoming ineffective. This paper describes, in detail, Verification Evidence and Resilient Design In anticipation of Cybersecurity Threats (VERDICT), a language and framework to address cyber resiliency. When we use the term resiliency, we mean hardening a system such that it anticipates and withstands attacks. VERDICT analyzes a system in the face of cyber threats and recommends design improvements that can be applied early in the system engineering process. This is done in two steps: (1) Analyzing at the system architectural level, with respect to cyber and safety requirements and (2) by analyzing at the component behavioral level, with respect to a set of cyber-resiliency properties. The framework consists of three parts: (1) Model-Based Architectural Analysis and Synthesis (MBAAS); (2) Assurance Case Fragments Generation (ACFG); and (3) Cyber Resiliency Verifier (CRV). The VERDICT language is an Architecture Analysis and Design Language (AADL) annex for modeling the safety and security aspects of a system's architecture. MBAAS performs probabilistic analyses, suggests defenses to mitigate attacks, and generates attack-defense trees and fault trees as evidence of resiliency and safety. It can also synthesize optimal defense solutions—with respect to implementation costs. In addition, ACFG assembles MBAAS evidence into goal structuring notation for certification purposes. CRV analyzes behavioral aspects of the system (i.e., the design model)—modeled using the Assume-Guarantee Reasoning Environment (AGREE) annex and checked against cyber resiliency properties using the Kind 2 model checker. When a property is proved or disproved, a minimal set of vital system components responsible for the proof/disproof are identified. CRV also provides rich and localized diagnostics so the user can quickly identify problems and fix the design model. This paper describes the VERDICT language and each part of the framework in detail and includes a case study to demonstrate the effectiveness of VERDICT—in this case, a delivery drone.

Keywords: cyber resilient systems engineering; model-based architectural analysis and synthesis; assurance case generation; cyber resiliency verification; formal analysis

Citation: Meng, B.; Larraz, D.; Siu, K.; Moitra, A.; Interrante, J.; Smith, W.; Paul, S.; Prince, D.; Herencia-Zapana, H.; Arif, M.F.; et al. VERDICT: A Language and Framework for Engineering Cyber Resilient and Safe System . *Systems* **2021**, *9*, 18. https://doi.org/10.3390/systems9010018

Received: 6 January 2021
Accepted: 20 February 2021
Published: 3 March 2021

1. Introduction

Cyber-physical systems (CPS) are systems built upon an integration of networking, control, computation, and physical components. These systems are prevalent in the areas of medicine, transportation, energy, and finance, among others. They have significantly transformed and improved the quality of people's lives. However, the ever-increasing complexity of cyber-physical systems drives the necessity for assurance of critical infrastructure and embedded systems. Traditional mechanisms to secure cyber-physical systems—such as using cyber best practices, adapting mechanisms from information technology (IT) systems, and penetration testing followed by patching—have proven to be generally ineffective. Failure of such systems could cause not only financial loss, but potentially loss of life. Therefore, ensuring the safety and security of CPS is paramount to other development tasks. Safety focuses on unintentional events happening in the system that could cause damage to its operating environment; security concentrates on threats from outside the system, such as malicious parties. The goal of ensuring the security of CPS is to protect confidentiality, integrity, and availability of information in the system. In the context of information security, confidentiality means that the information is only accessible to those who are authorized to access it; integrity means that no unauthorized modification can be made to the information or the system that handles the information; availability means that information is available to authorized users when needed. Likewise, the goal of ensuring the safety of a CPS is to prevent undesired events, such as loss of availability or loss of integrity from system components that cause system dysfunction and adversely impact operation. Thus, the goal of ensuring the safety of a system requires reduction in the severity of the consequences of those events. According to ARP-4761, the aerospace guideline for conducting safety assessments, safety is more concerned with availability and integrity of information flow. Cyber resiliency means the system continues to execute its mission and deliver the intended outcome even under cyber-attack. To develop systems that are both cyber resilient and safe, it is advantageous to address security and safety concerns from the beginning of the system development cycle.

The main contributions of this paper encompass (I) a language to model cyber and safety aspects of system architecture and (II) a framework to identify system vulnerabilities, and suggest safety and security mitigations in the early design phase. VERDICT [1] aims to analyze system architectures with respect to cyber and safety requirements in the face of cyber-attacks, and perform system behavior analysis against cyber-resiliency properties. The framework consists of three major functionalities: (1) Model-Based Architectural Analysis and Synthesis (MBAAS) (2) Assurance Case Fragment Generation (ACFG) and (3) Cyber Resiliency Verification (CRV).

This paper is structured as follows: Section 2 discusses the related work; Section 3 describes the background; Section 4 describes the VERDICT language; Section 5 details the framework with a case study example—a delivery drone—to demonstrate the capabilities and effectiveness of the VERDICT tool suite. This is followed by the conclusion in Section 6.

2. Related Work

Current cyber resiliency techniques generally fall into three leading paradigms. The first paradigm focuses on repositories in which to group and categorize security threats. For instance, [2] developing an attack taxonomy by analyzing Common Vulnerabilities and Exposures (CVE) data [3], using heuristics to match CVE records to a whitelist and blacklist of keywords. CVE data identifies countermeasures against software attacks and groups them into nine categories. An important difference in the described work is the use of Common Attack Pattern Enumeration and Classification (CAPEC); CAPEC is more suitable than CVE, as it includes meta-level attack patterns that can be considered at a systems architecture-level agnostic of the specific technical implementations. The second paradigm focuses on the manual construction of predictive system attack analyses, such as attack/defense trees, from local attacks identified in the architectural model of the system. This approach is typically compositional, meaning that system-level attack/defense analy-

ses can be generated from the component-level and topology of the system. Techniques and tools based upon this paradigm include attack/defense trees [4]. An Integrated Environment for Synthesizing Attack Trees (ATSyRA) is a tooling environment to automatically synthesize attack trees of a system under study. ATSyRA allows advanced editors to specify high-level descriptions of a system, high-level actions to structure the tree, and ways to interactively refine the synthesis [5]. An important difference in the described work is the use of the National Institute of Standards and Technology (NIST) defense provided by the VERDICT tool when the likelihood of attack does not meet the mission requirement. The third paradigm focuses on automatically analyzing potential attacks in a system model—typically represented as a state machine—using formal verification techniques such as model-checking. This generally works by injecting attacks into a formal specification of a system and studying the effects of attacks on the system behavior. The results are then used by model-checking tools to verify whether system dependability requirements are being satisfied or violations of the requirements exist in normal or attacks conditions. Techniques in this category include Probabilistic Symbolic Model Checker (PRISM)—a tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior [6]. It has been used to analyze systems such as denial-of-service security threats [7], Kaminsky Domain Name Service cache-poisoning attack [8]. Another tool is Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance (SOTERIA[1]), a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. More specifically, SOTERIA automatically extracts a state model from a SmartThings IoT app and applies model checking to find property violations. An important difference in the described work is that the VERDICT tool checks cyber-resiliency properties using a model checker. And when a property is proved or disproved, a minimal set of vital system components that is responsible for the proof or disproof is provided. To our knowledge, our reduction of this minimization problem to Max Satisfiability Modulo Theories (MaxSMT), which is sub-case of Optimization Modulo Theories [11], and our use of MaxSMT technology is novel. The framework also provides rich and localized diagnostic information feedback for users to quickly identify and fix problems with the model.

The various fault tree construction techniques can be partitioned into two classes. The first class includes tools for manual construction of fault trees. Given a tree, these tools perform various safety analyses, such as minimal cutsets and quantitative analysis. Examples of such tools are OpenFTA and Windchill FTA. The difference in the described work is that VERDICT does not require a tree as input, which is the second class of techniques: those that generate fault trees from models. Comparative examples include HiP-HOPS [12], AltaRica [13,14], and Error Modeling (EMV2) Annex [15,16]. HiP-HOPS is an add-on tool for Simulink or Sim-X models for reliability annotations, which are used to automatically generate fault trees and Failure Mode Effect Analyses (FMEAs). Minimal cutsets are then generated from fault trees. AltaRica is a high-level modeling language for specifying the behavior of systems when faults occur. It automatically generates a static fault tree and uses a model checker to reason over dynamic properties of the system expressed in Linear Temporal Logic (LTL). The third class is the AADL annex EMV2, which adds model error type, error propagations, composite error behavior, and component error behavior to an AADL model. Differences in the described work are that: EMV2 essentially requires users to provide a fault tree, as it requires a composite error behavior that specifies the logic in a fault tree; AltaRica rejects models with loops even if there are no cycles in the failure propagation, whereas VERDICT can resolve loops in a model.

The various assurance case techniques generally fall into two leading paradigms. The first paradigm focuses on claims, evidences, and structured arguments which justify how the evidences can satisfy the claims. Techniques in this category include GSN patterns to overcome challenges on how to integrate and harmonize critical issues on safety in

[1] Note that Soteria [9] is different from the SOTERIA [10] incorporated in our framework, described later.

addition to security for their systems [17–21]. The second paradigm focuses on building an argument attached to AADL models. The assurance cases are automatically updated as the architecture model evolves, and they never fall out of sync with the model [22,23]. An important difference in the described work is that the VERDICT tool automatically builds a GSN assurance case for safety and security using the evidence that is generates. This GSN assurance case is attached to AADL model, mission, safety, and security requirements.

3. Background

3.1. Model-Based System Engineering

Model-Based Engineering (MBE) includes paradigms such as Model-Driven Engineering (MDE), Model-Driven Development (MDD), Model-Based Systems Engineering (MBSE), and Model-Based Software Engineering. MBSE captures aspects of MBE such as system architectural design, behavioral analysis, and requirements traceability in the Systems Engineering domain. In MBE, different types of models such as a System Concept of Operations Model, System Architecture Model, and Reliability Model are used in different phases of the Software Development Lifecycle (SDLC) [24] to help formalize the approaches used in each phase. For example, the system architectural model focuses on system requirements, behavior, structure, interconnections, and properties, and is captured during the design phase of SDLC using modeling languages or tools such as Sys-ML, AADL, CAD, and UML. The captured model is then used as a blueprint during the next phases of the SDLC, thereby driving a requirements-based development process. MBSE has several advantages including the ability to foresee impacts of design changes and identify critical architectural components and dependencies. Furthermore, in MBSE, analysis of system architectural models can help in detecting bugs earlier in the SDLC, thereby reducing overall debugging and maintenance cost incurred at later phases.

3.2. Architecture Analysis and Design Language (AADL)

AADL [25] is an architecture analysis and design language standardized by the Society of Automobile Engineers (SAE) for modeling and analyzing real-time, safety-critical embedded systems. It is widely used in model-based engineering of software and hardware architectures. AADL is supported by several backend tools for different analysis including: scheduling, safety and reliability, model checking, and code generation. An AADL model consists of software components (data, thread, thread group, subprogram, process), execution platform components (memory, bus, processor, device, virtual processor, virtual bus) and hybrid components (system). Most components may contain nested subcomponents that together comprise a hierarchical model. The interface of a component is defined through the feature section of a component type. A component may have zero or more implementations, where a component implementation defines internal subcomponents, connections among them, properties of each subcomponent, and so on. Components must be instantiated as subcomponents of other component implementations to be meaningful in a hierarchical architecture model. Communications among components are defined in the connections block of a component implementation. AADL provides extensible language interface for user-defined annexes for different analysis such as Error Model Annex v2.

3.3. AGREE

The AADL language allows one to embed domain-specific languages for stating goals, behaviors, and other aspects of the system (e.g., security, safety) through the annex mechanism. One such annex is the AGREE annex [26] that enables one to encode finer-grained details about a system component's behavior in the form of an assume-guarantee contract. Such a contract for a given component in general asserts that the component's behavior guarantees certain formal properties when certain assumptions on the component's inputs are met. Such a specification mechanism of a system's behavior enables one to specify a system hierarchically and verify its fine-grained global guarantees compositionally. The semantics of the AGREE language can be defined with respect to an enhanced Lustre

language [27], which is a synchronous dataflow language with extensions to write contracts of components of a system for compositional analysis. In our context, the AGREE language is heavily used for specifying the system behavior necessary for the analysis by the CRV functionality.

4. The VERDICT Language

The VERDICT language is developed as an AADL annex called VERDICT annex for modeling the safety and security aspects of system architecture. It enables users to encode mission, safety and cyber requirements, and vulnerabilities propagations in the system via cyber relations, safety relations, and events. This information will be used for attack-defense tree and fault tree analysis of the system. A description of the syntax and semantics of the VERDICT language is as follows.

Syntax and Semantics

The language definitions of "MissionReq", "CyberReq", "SafetyReq", "CyberRel", "SafetyRel" and "Event" are shown in Table 1, where "MissionReq" stands for mission requirement; "CyberReq" stands for cyber requirement; "CyberRel" stands for cyber relations; similarly for "SafetyReq" and "SafetyRel". To achieve a collective security and safety goal of the system, cyber and safety requirements are grouped by mission requirements defined in the field "reqs" in a "MissionReq" block. For Boolean typed fields, the Boolean expression is constructed using logical operators: and, or, and not applied on atomic formulas. An atomic formula is constructed in terms of confidentiality, integrity and availability of ports of components using an operator ":". The semantics of "port: V" is that the port is V-concerned, where V is one of [C; I; A], and C, I and A stands for Confidentiality, Integrity and Availability respectively. An example of such formula is "input1: C or (input2: I and input3: A)". The "condition" field of "CyberReq" describes a failure condition of the system. The "severity" field of "CyberReq" is an enumerated type with values of [None; Minor; Major; Hazardous; Catastrophic]. It describes the severity of failure condition on its operating environment. Each severity value corresponds to a likelihood with None = 1e-0; Minor = 1e-3; Major = 1e-5; Hazardous = 1e-7; and Catastrophic = 1e-9. Security vulnerability propagations are encoded in "CyberRel" blocks with "inputs" being a Boolean logical expression in terms of confidentiality, integrity and availability of inputs of a component, and "output" being a Boolean logical expression in terms of confidentiality, integrity and availability of a single output of a component. The semantics of "CyberRel" is that cyber confidentiality, integrity, and availability of outputs specified in "output" is affected by the vulnerabilities defined in "inputs". The semantics of "CyberReq" is that given the vulnerability propagations defined in "CyberRel" in the system and a system architecture, the calculated likelihood of the successful attacks to "condition" shall not exceed the likelihood corresponding to the specified "severity".

Table 1. The language definition for *MissionReq*, *CyberReq*, *CyberRel*, *SafetyReq*, *SafetyRel* and *Event* with the left column being the field name and the right column being the field type. The range of scientific number is [0, 1].

MissionReq		**CyberReq**		**CyberRel**	
id	String	id	String	id	String
reqs	List of String	condition	Boolean	inputs	Boolean
comment	String	severity	Enumerate	output	Boolean
description	String	description	String	description	String
		cia	Enumerate		
SafetyReq		**SafetyRel**		**Event**	
id	String	id	String	id	String
condition	Boolean	faultSrc	Boolean	probability	Scientific Number
targetProbability	Scientific Number	output	Boolean	comment	String
description	String	description	String	description	String

A safety requirement is declared in a "*SafetyReq*" block. The "*targetProbability*" field of "*SafetyReq*" indicates the acceptable probability of failure of the system. Since safety is concerned about undesirable events happening in a system such as loss of availability, a block "*Event*" is introduced, and the probability of the event is defined in the field "*probability*". A safety vulnerability propagation is encoded in a "*SafetyRel*" block. The "*faultSrc*" is a logical Boolean expression in terms of happening of events and integrity and availability of inputs of a component. A predicate "*happens*" is introduced that is applied on an event "*id*". The "*output*" field is a logical Boolean expression in terms of integrity and availability of a single output of a component. The semantics of "*SafetyRel*" is that the failure of "output" is induced by the failure of "*faultSrc*". The "SafetyReq" states that given the "*SafetyRel*" for a system architecture and the "*condition*", the calculated probability of system failure shall not exceed the specified "*targetProbability*".

In AADL, "MissionReq", "CyberReq" and "SafetyReq" are only allowed to be associated with the top-most component type; whereas "CyberRel" and "SafetyRel" are only allowed to be associated with component types that are used as subcomponents.

5. The VERDICT Framework

The VERDICT framework is described below in detail, including the workflow, model construction process, and various features of the tool-suite.

5.1. VERDICT Workflow

An overview of the VERDICT workflow is illustrated in Figure 1. VERDICT is developed as a plugin to the Open Source AADL Tool Environment (OSATE). The user captures an architecture model using AADL that represents the high-level functional components of the system along with the data flow between them. Additional security and safety models are required to perform MBAAS and ACFG; similarly, additional behavior models are needed to perform CRV. The input models are translated into a VERDICT internal data model, which will be further translated into inputs for MBAAS, ACFG, and CRV, accordingly. From within the OSATE environment, the user invokes each one of these via a drop-down menu. Each functionality comes with a configuration panel allowing users to set parameters for analysis.

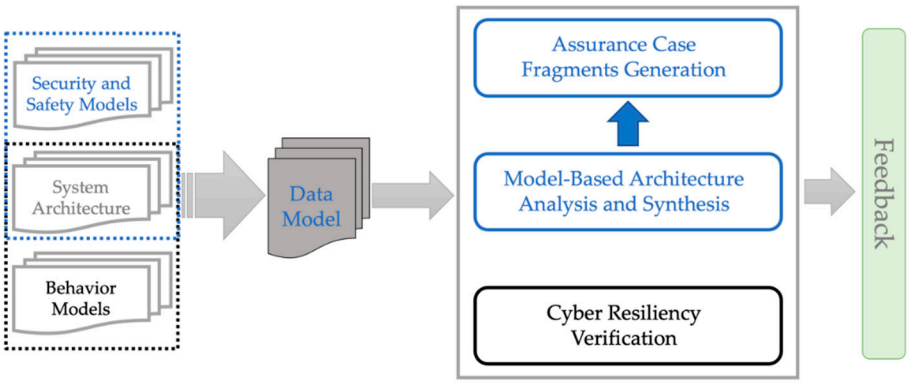


Figure 1. Overview of the VERDICT workflow.

For MBAAS and ACFG, the system engineer needs to identify and specify mission requirements, safety and cyber requirements, and cyber and safety relations of the system. The cyber and safety relations define how vulnerabilities propagate through the system. The cyber requirements of the system are defined in terms of *confidentiality*, *integrity* and *availability* outputs of the top-level system; the safety requirements are defined in terms of *integrity* and *availability* of outputs of the top-level system. For safety analysis, MBAAS performs fault-tree analysis to calculate the probability of system failure based on the error events and propagations defined in the system. For security analysis, MBAAS analyzes the architecture to identify cyber vulnerabilities and recommend defenses. These defenses are

recommendations to either improve the resiliency of existing components, such as control access and encrypt communication links, or recommendations to add new components to reduce dependence on a specific source of information. By running MBAAS, the designer is able to identify whether or not the system can achieve its goals due to subcomponent vulnerabilities. Specifically, MBAAS helps the designer identify such vulnerabilities and further suggests alternate design options. Implementation of defenses often has associated costs. Given the cost of each defense, MBAAS can synthesize an optimal solution with a minimal set of defenses with minimal implementation costs. In addition, the assurance case fragments generation capability can automatically assemble the MBAAS assurance evidence into Goal Structure Notation (GSN) form for certification purposes.

Once the architectural analysis is complete, VERDICT supports refinement of the architecture model with behavioral information using AGREE. The VERDICT CRV back-end tool performs a formal analysis of the updated model with respect to formal cyber properties to identify vulnerabilities to cyber threat effects. Specifically, CRV performs architectural analysis by focusing on the behavioral information of the system and its sub-components. The behavioral information (i.e., details of how inputs are transformed into outputs) can be expressed as "assume-guarantee contracts" in AGREE, along with the system architecture (defined in AADL), goals, behavioral information, and information related to threats are fed as inputs to CRV. CRV analyzes and reports if the goals/requirements are satisfiable even in the presence of threats. If the goals/requirements are satisfiable, CRV returns a merit assignment result (i.e., a set of components that played a vital role in goal satisfaction). If the requirements/goals are unsatisfiable, then CRV returns a blame assignment result (i.e., a set of components that played a vital role in requirement violation). The CRV capability provides an additional depth-of-analysis of a model that includes behavioral details of the architectural component models that help to identify design mistakes early in the development process. Once the CRV analysis is complete, the developer can create a detailed implementation. MBAAS and CRV work collaboratively to analyze the system design for resiliency. MBAAS analyzes the system design for resiliency using minimal information; CRV captures behavioral information to validate requirements that need such detailed behavioral information for their analysis.

5.2. Model Construction in VERDICT

We will use a delivery drone to demonstrate how to construct an AADL model that is analyzable by VERDICT. The AADL source code is publicly available on the Github repository [28]. The drone system architecture is shown in Figure 2. The drone is part of the delivery system that consists of a van with packages to be delivered and one or more of the delivery drones. After the van arrives at a location that is close to multiple delivery sites, the delivery drones are initialized with their current position, delivery location, and the package to be delivered is loaded. After a delivery drone is launched, it uses the inputs from the GPS and IMU to navigate to the delivery location. When the drone reaches the delivery location, it uses its Camera to capture an image of the receiving site to confirm that it is free of obstacles and it is safe for the package to be dropped off. For a high-value package, the Delivery Planner will use the Radio component to get confirmation from the operator in the van. If there are no obstacles on the receiving site and confirmation (if needed) is received from the operator, then the Delivery Planner will activate the Delivery Item Mechanism to drop off the package. The delivery drone also needs to avoid certain airspace, for example airports, schools and government buildings. The Global Positioning System (GPS) and Inertial Measurement Unit (IMU) are subcomponents of Guidance, Navigation and Control (GNC) to illustrate model hierarchy.

5.2.1. Modeling System Architecture in AADL

The VERDICT tool-suite supports a core subset of AADL constructs shown in Table 2, which can be extended as needed.

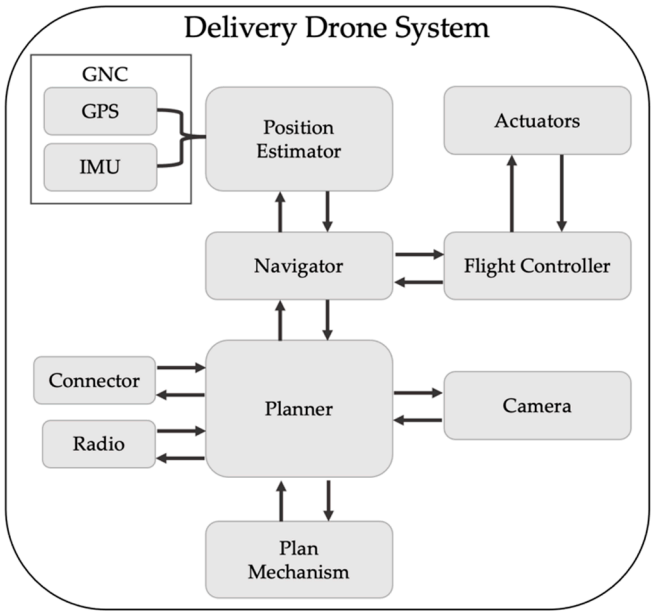


Figure 2. The Delivery Drone System Architecture.

Table 2. The subset of AADL language constructs supported by VERDICT.

AADL Construct	Feature
system/abstract/device/process/thread type	**ports**: in/out/in out provides/requires data access
system/abstract/device/process/thread implementation	**subcomponents**: property associations **connections**: unidirectional, bidirectional, property associations
property set	**property**: Boolean/string/integer/enumerated type with default values

To analyze the system design of the delivery drone using VERDICT, the system designer will need to model the system architecture in AADL, specify the meta-level and defense properties of the system and its subcomponents, and encode safety and cyber requirements at the system and subcomponent level in VERDICT annex. Since we will model only a conceptual view of the drone system, each subcomponent will be modeled as either a system type or a system implementation.

Figure 3 shows a snippet of the AADL code for the top-level delivery drone system type and its implementation (some implementation and connection details are omitted for the sake of space). The system type has a feature section with data port (*in*ports and *out*ports) declarations. The system implementation encompasses various subcomponent instances and internal connections. Each subcomponent is an instance of a system type or implementation.

5.2.2. VERDICT Properties

VERDICT analyzes a system architecture model based on a set of built-in meta-level properties. The properties reflect various design properties of the system. The complete set of VERDICT properties is summarized on the Wikipedia page of the VERDICT repository[2]. There are six types of VERDICT properties: (1) mandatory properties; (2) port properties; (3) connection properties; (4) component properties; (5) connection cyber defense proper-

[2] https://github.com/ge-high-assurance/VERDICT/wiki/Model-Building#declaring-and-setting-verdict-propert (accessed on 2 March 2021).

ties; and (6) component cyber defense properties. Mandatory properties are those that must be set for every system for VERDICT to perform analysis. If no value is set, default values (that correspond to least secure settings) are set for those properties. The only port property is probe, and it signifies to CRV that the port is not part of the original architecture, but that it is included just for the convenience of CRV's reasoning. When a system is modeled hierarchically, a probe port allows CRV to look inside a component's behavior without needing to expose the behavior completely at a higher level. Connection properties are those that apply to connections but are optional for performing analysis. If not applied, the VERDICT tool suite will default to the conservative values as defined. Component properties are those that apply to systems but are optional for performing analysis. If not applied, VERDICT tools will default to the conservative values as defined. The connection and component defense properties are of enumerated integer type [0; 3; 5; 7; 9], which describes the implementation rigor of defense, with 0 being the lowest rigor and 9 being the highest rigor. VERDICT properties are set in the "feature (port)" section of component type and "subcomponents" and "connections" sections of the component implementation, accordingly. The AADL code snippet in Figure 3 shows the port, component, and connection property associations in the delivery drone model in blue color.

```
system DeliveryDroneSystem
    features
      bus1: in data port Data_Types::InputBus.impl;
      comm1: in data port Data_Types::RadioResponse.impl;
      comm2: out data port Base_Types::Boolean;
      bus2: out data port Base_Types::Boolean;
      radio_cmd: out data port Base_Types::Boolean;
      delivery_status: out data port Data_Types::DeliveryStatus;
      actuation_out: out data port Base_Types::Boolean;
      probe_init_mode: out data port Base_Types::Boolean;
      probe_fly_cmd: out data port Base_Types::Boolean {
          CASE_Consolidated_Properties::probe => true; };
    annex agree {** … **};
    annex verdict {** … **};
end DeliveryDroneSystem;

system implementation DeliveryDroneSystem.Impl
    subcomponents
      gnc: system GNC::GNC.Impl;
      radio: system Radio;
      positionEstimator: system PositionEstimator;
      navigation: system Navigation;
      fc: system FlightControl;
      actuation: system Actuation;
      deliveryPlanner: system DeliveryPlanner;
      deliveryItemMechanism: system DeliveryItemMechanism{
          CASE_Consolidated_Properties::componentType => Hardware;
          CASE_Consolidated_Properties::hasSensitiveInfo => true;
          CASE_Consolidated_Properties::supplyChainSecurity => 7;};
      camera: system Camera;
      connector: system Connector;
    connections
      c1: port positionEstimator.est_pos -> navigation.est_pos{
          CASE_Consolidated_Properties::connectionType => Trusted;
          CASE_Consolidated_Properties::encryptedTransmission => 5;};
      c1b: port deliveryPlanner.launch_pos -> gnc.launch_pos;
      c2: port navigation.move -> fc.move;
      c3: port fc.fc_state -> navigation.flight_control_state;
end DeliveryDroneSystem.Impl;
```

Figure 3. An AADL code snippet of the top-level delivery drone system type and implementation declarations with port, component and connection properties annotations.

5.2.3. Modeling Cyber-Security and Safety Aspects of System Architecture in VERDICT Annex

The cyber-security and safety aspects of the system are modeled using the VERDICT annex introduced in Section 3. Cyber, safety, and mission requirements may only be declared in a VERDICT annex within the top-level system type. Cyber and safety require-

ments can be aggregated and associated to a particular mission requirement. Cyber and safety relations may only be declared within a subcomponent system type. They describe the vulnerability and failure flow, respectively, between the inputs and outputs of an individual component within the system. Safety relations model how faults or (erroneous) events or inputs affect the Integrity (I) and Availability (A) of the outputs. Cyber relations model how cyber-attacks affect the Confidentiality (C), Integrity (I), Availability (A) of the outputs.

- Mission Requirements

Figure 4 shows an example of a mission requirement in the VERDICT annex. Cyber Requirements "CyberReq01"and "CyberReq02" and a safety requirement "SafetyReq01" are aggregated to support the mission requirement "MReq01".

```
MissionReq {
    id ="MReq01"
    description = "Deliver a package to the intended
                   location."
    reqs = "CyberReq01","CyberReq02","SafetyReq01"
};
```

Figure 4. A mission requirement for the delivery drone.

- Cyber Requirements & Relations

Figure 5 shows an example for defining a cyber requirement in VERDICT annex. Cyber requirements must only be declared at the topmost system level of the AADL project. It is recommended to enter a message of the form "The loss of <Confidentiality, Integrity, Availability >of the subject variable shall be <None, Minor, Major, Hazardous or Catastrophic> in the "description" field. For example—"The loss of Integrity of the estimated position signal input of the Navigator shall be Hazardous". The following list shows the acceptable likelihood of successful attack value for each of the severity levels: Catastrophic = 1e-9; Hazardous = 1e-7; Major = 1e-5; Minor = 1e-3; None = 1e-0.

```
CyberReq {
    id = "CyberReq01"
    description = "The drone shall be resilient to loss
                   of ability to deliver a package to the
                   appropriate consumer location"
    condition = actuation_out:I or actuation_out:A
                or delivery_status:I or delivery_status:A
    cia = I
    severity = Hazardous
};
```

Figure 5. A cyber requirement for the delivery drone.

Cyber relations are used to map component vulnerability of inputs to outputs. VERDICT requires the user to declare the cyber relations for each component in the AADL model. Cyber relations represent the relationship of the input and output signals of a component. Cyber relations are defined in the declaration section of the component type in AADL using the VERDICT annex. Figure 6 shows two cyber relations for a DeliveryItemMechanism component. The cyber relations specify that the integrity and availability of *delivery_status_out* is impacted by the integrity and availability of *delivery_cmd_in*, respectively.

```
system DeliveryItemMechanism
    features
      -- inputs
      delivery_cmd_in: in data port Data_Types::PackageDeliveryCommand;
      -- outputs
      delivery_status_out: out data port Data_Types::DeliveryStatus;
      package_is_secure: out data port Base_Types::Boolean;

annex verdict {**
    CyberRel "delivery_status_out_I" = delivery_cmd_in:I => delivery_status_out:I;
    CyberRel "delivery_status_out_A" = delivery_cmd_in:A => delivery_status_out:A;
**};
end DeliveryItemMechanism;
```

Figure 6. Cyber relations for the *DeliveryItemMechanism*.

- Safety Requirements, Error Events, & Relations

Figure 7 shows an example of a safety requirement in the VERDICT annex. Safety requirements must only be declared at the top-most system level of the AADL project.

```
SafetyReq {
    id = "SafetyReq01"
    description = "Loss of actuation shall
                   be less than 1e-7 pfh"
    condition = actuation_out:A
    targetProbability = 1e-07
};
```

Figure 7. A Safety Requirement for the delivery drone.

Figure 8 shows a loss of availability (LOA) error event and a safety relation for the *DevliveryItemMechanism* component in the VERDICT annex. The probability of loss of availability event is 1.0e-8. The safety relation describes that the availability of *delivery_status_out* is affected by the occurrence of LOA event of the component or if the availability of *delivery_cmd_in* is affected.

```
system DeliveryItemMechanism
    features
      -- inputs
      delivery_cmd_in: in data port  Data_Types::PackageDeliveryCommand;
      -- outputs
      delivery_status_out: out data port Data_Types::DeliveryStatus;
      package_is_secure: out data port Base_Types::Boolean;
annex verdict {**
    Event {
      id = "loa_event"
      probability = 1.0e-8
      comment = "loss of availability of the DeliveryItemMechanism"
      description = "LOA"
    }
    SafetyRel "delivery_status_out_LOA" = happens("loa_event") or delivery_cmd_in:A
                                          => delivery_status_out:A;
**};
end DeliveryItemMechanism;
```

Figure 8. An error event and a safety relation for the *DeliveryItemMechanism* component.

5.2.4. Modeling System Behavior and Cyber Properties in AGREE Annex

Once the system architecture and the associated meta-data, which are necessary for making the system-under-test amenable to the MBAAS' architecture-level security and safety analysis, have been filled, the next step is to add the system's behavior-level information and guarantees. This is critical for making the system ready for a more fine-grained analysis performed by the CRV, compared to the conservative analysis performed by MBAAS. This behavior-level information includes two types of information, namely, the assume-guarantee style contracts for each component as well as the formal functional

property that the system should verifiably maintain even under attack. For explaining the process of populating the architectural model of the system with behavioral information, we will use the delivery-drone model presented in Figure 2.

Specifying behavioral information of the system: In the first step, we need to add behavioral information for the components in the architectural model of our delivery drone system. This will include abstractly specifying the intended functionality of that component. The component-level behavior is abstract in the sense that it does not necessarily capture all the fine-grained behavioral details of the component in question. As an example, for the analysis performed by CRV of an unmanned aerial vehicle (UAV) system, it may be sufficient to just model that the UAV moves from one waypoint to another without capturing details regarding how it precisely navigates. Abstractly capturing the component-level behavior also allows CRV to take advantage of automated formal reasoning techniques, such as model checking, while avoiding scalability challenges often incurred by such formal techniques (i.e., state-space explosion). For behavioral specification, we use AGREE contracts that allow us to choose a level of abstraction in the specification that is sufficient to prove the system-level cyber-resilience property (as opposed to provide a complete and detailed specification). At a high-level, this functional behavior for each component simply expresses temporal constraints between the values of *out*ports and the value of *in*ports. Sometimes these constraints may need to be expressed also in terms of the internal state of a component. In that case, relevant information on the internal states may be provided using virtual output ports that we refer to as *probes*. Note, these virtual output ports are introduced for reasoning purposes and do not need to be realized in real-life deployments of the system.

The overall system is assumed to run on a universal base clock that represents the smallest time span the system can distinguish. Note that the restriction to a synchronous model of computation is intentional, as this model is better suited to translation into a programming language, as it more naturally matches the behavior of a computer program. Moreover, usually this restriction can be overcome by faithfully simulating asynchronous systems in the synchronous model in a variety of ways.

For review purposes of prior information, the *DeliveryItemMechanism* component in our Delivery Drone model has the following *in*ports and *out*ports:

- *delivery_cmd_in*: *in*port representing a command received by the delivery mechanism
- *delivery_status_out*: *out*port representing the current status of the delivery
- *package_is_secure*: *out*port representing whether the package is secure or not

Specifying the relation between *out*ports and *in*ports (and/or the internal states) depends on the level of abstraction we want to have for this component which eventually hinges on the properties we want to verify. However, the rule of thumb is to stay at the highest level of abstraction first and then try to verify the properties. If we need to specify more about this component to prove the desired system-level properties, we can refine its specification later. Assuming for the *DeliveryItemMechanism* component that at least the following information is known:

1. The component should accept two commands: *release a package* and *abort a delivery*.
2. The component must be in one of these four possible states: the delivery has *not started*, it is *in progress*, it has been *completed*, or it has *failed*.
3. Initially the delivery has *not started*.
4. If a delivery command is issued, the delivery status must become different from *not started*.
5. If no command is issued or an *abort* command is received, then delivery status gets reset to *not started*.

This is a minimal level of information one can anticipate about the expected behavior of the *DeliveryItemMechanism* component. For formalizing (1) and (2) in AADL, we define two enumeration types shown in Figure 9:

```
data PackageDeliveryCommand
    properties
        Data_Model::Data_Representation => Enum;
        Data_Model::Enumerators =>
            ("NO_OPERATION", "RELEASE_PACKAGE", "ABORT_DELIVERY");
end PackageDeliveryCommand;

data DeliveryStatus
    properties
        Data_Model::Data_Representation => Enum;
        Data_Model::Enumerators =>
            ("NOT_STARTED", "IN_PROGRESS", "COMPLETED", "FAILED");
end DeliveryStatus;
```

Figure 9. Enumerate type definitions for formalizing requirement (1) and (2).

Notice that the enumeration type *PackageDeliveryCommand* has a value *NO_OPERATION*, that allows us to distinguish the case where no command is issued, which is required by the system (5). Alternatively, one could use event ports to implicitly model the case where no signal is present and keep the enumeration type with only two values. This modeling choice relies on whether an explicit value is needed to reason about other parts of the system.

To state aspect (3) formally, we add the following guarantee stating that the delivery status is equal to *NOT_STARTED* initially, as it is safe to assume that the system starts in a proper initial state. We achieve this by using an auxiliary AGREE operator *InitiallyX* whose definition is shown in Figure 10.

```
guarantee "Initially, delivery status is NOT_STARTED":
  Agree_Nodes::InitiallyX(delivery_status_out = Agree_Constants::NOT_STARTED_STATUS);

node InitiallyX(X: bool) returns (Y: bool);
let
    Y = X -> true;
tel;
```

Figure 10. Definition of *InitiallyX* operator in AGREE annex.

The *InitiallyX* operator accepts a Boolean expression as input and evaluates to true only if the Boolean expression is true initially. Note that, the infix initialization operator -> is natively supported in the AGREE language. An expression of the form e1 -> e2 evaluates to the value of expression e1 initially and to the value of e2 at all later steps of the system's execution. For formally capturing aspect (4), we add the following guarantee, in which *release_cmd* is an auxiliary variable that we have introduced for readability purposes and is shown in Figure 11.

```
guarantee "If delivery command is issued, delivery status is different from NOT_STARTED":
  true -> (release_cmd => delivery_status_out <> Agree_Constants::NOT_STARTED_STATUS);

eq release_cmd: bool = (delivery_cmd_in = Agree_Constants::RELEASE_PACKAGE_CMD);
eq abort_cmd: bool = (delivery_cmd_in = Agree_Constants::ABORT_DELIVERY_CMD);
```

Figure 11. Guarantee definition in AGREE annex for formalizing aspect (4).

Similarly, we capture aspect point (5) with the following guarantee that uses two new auxiliary variables shown in Figure 12.

```
guarantee "if no op or abort command have received then delivery status gets re-started":
  true -> (no_op_cmd or abort_cmd => (delivery_status_out = Agree_Constants::NOT_STARTED_STATUS));

eq abort_cmd: bool = (delivery_cmd_in = Agree_Constants::ABORT_DELIVERY_CMD);

eq no_op_cmd: bool = (delivery_cmd_in = Agree_Constants::NO_OPERATION_CMD);
```

Figure 12. Guarantee with auxiliary variables definitions in AGREE annex for formalizing aspect point (5).

So far, we have only added constraints about the delivery_status_out *out*port. Any value that satisfies those constraints will be considered valid during the analysis performed by CRV. This also means that since we have not added any constraints regarding the signal *package_is_secure*, CRV is free to assume that it can take any value allowed by its type. We want to emphasize that there are situations when one needs to add more detail to the specification to capture the behavior of a component more precisely for a finer-grained analysis. Such a situation can be observed in the specification of the *DeliveryPlanner* component. The AGREE specification of this component relies on an abstract representation of the component internal states (using the notion of modes) to specify its functionality.

Specifying desired formal property of the system: The next step is to review the list of safety functional requirements for the system that may affect its integrity and formalize them as formal cyber-resiliency properties in the AGREE language. For instance, consider that we are given the following cyber-requirement for DeliveryDroneSystem that regulates when a package could be released: *The drone never initiates packet release to an off-limits location* (labeled as P7). To formalize the cyber-requirement P7, we must first identify the components and ports of the system that are relevant to the description of the property. In our example, the *DeliveryPlanner* is the component that issues the command to release a package by setting the output port *delivery_cmd* to *RELEASE_PACKAGE*, and the *DeliveryItemMechanism* is the component that receives the command and proceeds with the delivery. Moreover, to know where the drone should release the package, the *DeliveryPlanner* reads the delivery location from the input port *bus1* through the Connector component when the drone is in the van, and then it passes this value to the Navigation component. In addition, we also need to know when a location is off-limits. For that, we must define a new predicate over locations that evaluates to true only if the location is within a restricted area. The specific definition of the predicate is irrelevant to the analysis and could have been left abstract as an uninterpreted predicate. However, AGREE does not allow the user to declare an uninterpreted predicate, so one needs to fix an arbitrary definition. The following is an example definition of the predicate in which X_LOW, X_HIGH, Y_LOW, and Y_HIGH are fixed constants of type real (Figure 13).

```
const X_LOW: real = 30.0; const X_HIGH: real = 50.0;
const Y_LOW: real = -83.0; const Y_HIGH: real = -63.0;

node InRestrictedArea(p: Data_Types::Position.impl) returns (f: bool);
let
    f = (X_LOW <= p.x and p.x <= X_HIGH) and
        (Y_LOW <= p.y and p.y <= Y_HIGH);
tel;
```

Figure 13. Definition of *InRestrictedArea* in AGREE annex.

Once we have defined the predicate, we can express the cyber-property with the guarantee in Figure 14, in which *started* (defined below) is an auxiliary predicate which returns true if and only the delivery has started.

```
eq started: bool = (delivery_status <> Agree_Constants::NOT_STARTED_STATUS);

guarantee "P7: The drone never initiates packet release to an off-limits location":
    started => not Agree_Nodes::InRestrictedArea(probe_delivery_location);
```

Figure 14. Formal property P7 defined in AGREE guarantee.

The above guarantee needs access to the *delivery_location* information, which is a problem. Because the *delivery_location* port is not accessible from the Navigation interface. For these cases, the interface of Navigation and the DeliveryDroneSystem can be extended with a probe signal which is not part of the actual architecture of the system. This will only be used for specification and verification purposes. To identify the new port as a probe, the user can set the VERDICT property "*probe*" to *true* for the new port as shown in Figure 15.

```
probe_delivery_location: out data port Data_Types::Position.impl
       {CASE_Consolidated_Properties::probe => true; };
```

Figure 15. Assigning value to "*probe*" property in AADL.

Note, it may be convenient to add a prefix to the port name to make it clear that the new port is a probe. This is only a modeling practice and is not required for the analysis. If the *delivery_status* port of the DeliveryItemMechanism component were also not accessible, we could proceed similarly by introducing a new probe signal for that information.

5.3. Model-Based Architectural Analysis and Synthesis (MBAAS)

The MBAAS tool consists of two functionalities: *analysis* and *synthesis*. The analysis takes in architecture models, mission, safety, and cyber-resiliency requirements, then generates fault and attack-defense trees with resiliency metrics. For the attack-defense tree analysis, the tool first leverages the security threat evaluation and mitigation (STEM) tool to identify possible CAPEC attacks and NIST-800-53 controls (defenses) to various components of the system based on annotated VERDICT properties. This information will be fed into SOTERIA++ to calculate the likelihood of successful attacks to determine the success or failure of cyber requirements. For safety analysis, the tool calculates the probability of system failure based on the error events, safety relations, and requirements. Conversely, the synthesis functionality focuses on cyber-security, using the attack-defense tree information along with cybersecurity requirements as inputs and generates defense properties associations that meet predefined resiliency design constraints. The MBAAS tool enables the system engineer to model components and then synthesize architectures that meet both safety (based on fault tree analysis) and security (based on attack-defense tree analysis) design goals.

5.3.1. Analysis of the Safety of System Architectures (SOTERIA++)

Safety and security have traditionally been handled separately—these topics are discussed in their own professional communities; businesses set up different departments to handle product safety and product security. Analyses have also traditionally been done separately and sequentially, where a system will normally first undergo safety analysis and then security. There is great benefit in analyzing safety and security simultaneously as they have undeniable interdependencies. For instance, consider an everyday example of an exit door: a locked door is secure but does not necessarily provide a safe egress in an emergency. Recognizing the ever-increasing complexity of cyber-physical systems, we developed a tool that reports on both safety and security of a system. In this section, we cover the foundation of safety analysis of system architectures.

MBAAS extends the original framework developed under a program called Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance (SOTERIA) [29]. The original work was limited to safety analysis. We extended the framework to include a security aspect (described in the next section), hence the name SOTERIA++. The overall philosophy is a compositional, model-based framework for better security and safety analysis, shifting the engineer's focus away from generating artifacts to expressing the properties that he wants in a system design. One such artifact for measuring safety is fault tree analysis (FTA). It is commonly used to examine the combination of undesired events that result in the system's inability to perform a mission under identified hazards. These hazards impact the loss of availability (inability to perform a function) and loss of integrity (inability to perform a function correctly). Fault trees—though powerful—are difficult to construct by hand. According to Banach, et al. [30], "The manual construction of fault trees relies on the ability of the safety engineer to understand and foresee the system behavior ... it is a time consuming and error-prone activity." In the product lifecycle, FTA is used in multiple stages of development. At the early stages, it is used to draft candidate architectures; at later stages, after a design is created, it is used to verify compliance with qualitative and quantitative safety objectives. Fault trees must be

kept up-to-date with inevitable updates to the architecture. The SOTERIA++ framework, which automatically generates the fault tree, supports such modifications and continual evaluation of complex system architectures. Furthermore, the SOTERIA++ framework is compositional in that fault propagation is defined at the component level, making it easy to maintain the architectural model.

Regarding the modeling of faults, there exists an Error Modeling Annex (EMV2) created by Julien Delange and Peter Feiler from Carnegie Mellon University to address the gap between safety requirements specification and systems engineering. An objective of According to EMV2's documentation, EMV2 is "to automate safety analysis methods by supporting them through analyzable architecture fault models". This is done by annotating the AADL model with hazard, fault propagation, failure modes and effects due to failures, as well as compositional fault behavior specifications. The steps to do safety analysis are well documented on GitHub and elsewhere [31–33] EMV2 is a very expressive, covering fault modeling in three levels: (1) error propagation—for each component, the user can specify the outgoing and incoming error types; (2) error behavior—for each component, the user can specify the error event, how, when coupled with incoming error propagations affect the error state, and under what conditions outgoing error propagations occur. These behaviors are specified via reusable error behavior state machines; and (3) compositional error behavior—for each component with subcomponents, the user can specify under what conditions in terms of subcomponent error states the component is in a particular error state. The composite error behavior specification must be consistent with the component error behavior. The expressivity of EMV2 give the tool the ability to generate many types of artifacts to support different certification documents, such as Functional Hazard Assessment (FHA), Fault Tree Analysis (FTA), and Failure Mode and Effect Analysis (FMEA).

We considered EMV2, and chose to not to integrate it into VERDICT. Instead, we developed our own representation in the VERDICT annex. In MBAAS, our focus was on generating fault trees. While EMV2 is very expressive, we found that only a small subset of the information required for EMV2 is needed. The information needed to generate fault trees are basically (1) a set of safety formulas for each subsystem, (2) a set of probabilities for those internal events of each subsystem, and (3) a set of safety requirements for the top-level system describing what must hold for that system's outputs. As an experiment we created a side-by-side comparison of the information needed for EMV2 to generate fault trees and the information needed for VERDICT to do the same. We created a simple AADL architecture with 3 sensors and a voter expressed in 31 lines of code. The architecture when annotated with EMV2 grew to 129 lines, while the model annotated with the VERDICT annex for safety was only 85 lines long. In addition to conciseness, we made an additional observation about EMV2 that had to do with usability: we always had to create a system that extends a previous system because it was not possible to express "properties" and "component error behavior" in the same code block. For these reasons, we opted not to adopt EMV2 into VERDICT.

In addition to conciseness and usability, another reason we opted for our SOTERIA framework is that EMV2 requires users to provide a composite error behavior at the system level which specifies "the logic in a fault tree" according to Delange, et al [17]. SOTERIA does not require such information from the user. It generates the fault tree from the safety relations specified at the component level as described in Section 5.2.3.

Here's an illustration of the safety analysis of the Delivery Drone System presented in Figure 2. The safety requirement presented in Figure 7 says, "*The loss of actuation shall be less than 1e-7 pfh.*" More formally, this means that on the Delivery Drone System the probability of losing the output *actuation_out* must be less than 1e-7. Having modeled compositionally the safety relations of each component of the Delivery Drone (such as the one illustrated in Figure 8 for the *DeliveryItemMechanism*) and having modeled the component connections of the Delivery Drone System, the fault tree for *SafetyReq01* is illustrated in Figure 16.

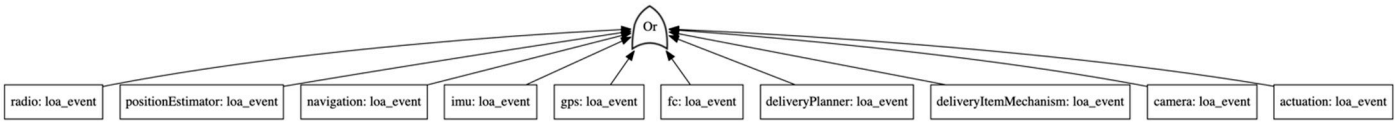


Figure 16. The fault tree for *SafetyReq01*.

5.3.2. Analysis and Synthesis of the Security for System Architectures

The assessment of the security of cyber-physical systems typically takes the form of an analysis of the risk that identified attacks will overcome mitigations within the system to cause undesired effects. A successful assessment can provide a blueprint for a secure architecture but must satisfy a few key concerns to be successful. First, a sufficient set of possible attacks must be identified to adequately cover the set of actual attacks that the system will experience when fielded. Second, appropriate mitigations must be determined and properly implemented to prevent these attacks from succeeding in causing the undesired effects. Third, the coverage of the attacks and sufficiency of the mitigations must be measured and communicated to the designer to compare the relative strength of alternative architectures and find the best solution. In addition to these significant elements of the analysis, synthesis can help the designer find the most secure architectural alternative faster. The following sections describe how MBAAS can be used to support each of these important aspects of a security assessment.

Security Threat Evaluation and Mitigation (STEM)

This approach depends on an appropriate library of threats, as controls will only be selected if they are useful in mitigating attacks that have a defined effect on the system under consideration. Threats are identified in terms of MITRE's CAPEC [34]. The use of a common threat library has two benefits. First, use of a common library incorporates a more diverse collection of attacks by sourcing information from a broad community. Second, a common library encourages threat information sharing using a common classification system and language. The controls are from NIST's Security and Privacy Controls 800-53 [35] and provide a set of security controls originally intended for federal information systems and organizations. However, these controls are now commonly applied to systems in domains outside of traditional information systems. Controls and enhancements are continually updated—there are 863 controls and enhancements as of revision four. Detailed mapping between the CAPECs, NIST controls and the architectural level component properties can be found in [36].

MITRE CAPEC defines a few classifications to attack patterns in their attack pattern hierarchy. Currently, not all levels can be readily translated to high-assurance embedded systems. These classifications include Category to generally break up the attacks by some common characteristics: Meta Attack Pattern defined as "a decidedly abstract characterization of a specific methodology or technique used in an attack"; Standard Attack Pattern, which is "focused on a specific methodology or technique used in an attack"; and Detailed Attack Pattern, which "provides a low level of detail, typically leveraging a specific technique and targeting a specific technology and expresses a complete execution flow".

Attacks defined at the Standard Attack Pattern and the Detailed Attack Pattern levels are tied to specific technologies and techniques used. As the basis of this hierarchy is a list of known enterprise IT vulnerabilities and weaknesses, these would not be readily translatable and often do not apply, and therefore were excluded from consideration in STEM. Instead, attacks at the Meta Attack Pattern level were used. In this way, STEM can consider abstract classes of attacks as they might be applied to embedded system architectures and create a rule set for the application of these attacks. For example, the Standard Attack Pattern "ICMP Flood" often does not apply to embedded systems, but the more abstract Meta Attack Pattern "Flooding" can be applied to incoming connections of any type. From this attack pattern, defenses can be prescribed to handle excessive traffic on the connection.

As an example, detailed CAPECs related to memory corruption are not included in the Meta Attack Pattern library. There is a total of 61 Meta Attack Pattern CAPECs, but not all of them are relevant to embedded systems of interest to STEM. 37 Meta Attack Pattern CAPECs that are relevant to an embedded system are identified have been incorporated into STEM. Mitigations are linked to CAPECs so that controls are only suggested if they are useful in mitigating attacks that have a defined effect on the system under consideration. The mapping between CAPECs, VERDICT cyber defense properties and NIST-800-53 controls is shown in Table A1 in Appendix A adapted from [36]. CAPECs that are mitigated by the same defense have been grouped together. Note that most of the CAPECs are mapped to a single defense, but some CAPECs require a combination of two defenses. Each defense is, in turn, mapped to a combination of more detailed NIST Controls. The detailed description of the NIST Controls can be found on the NIST website [37].

STEM uses a semantic model and rules to identify the vulnerabilities and defenses for an attack scenario. The model and rules in STEM are authored in Semantic Application Design Language (SADL) [38]. SADL is a controlled-English language that maps directly into the Web Ontology Language (OWL) [39]. It also contains additional constructs to support rules, queries, testing, debugging, and maintaining of knowledge bases. Besides being a language, SADL is also an integrated development environment (IDE) for building, viewing, exercising, and maintaining semantic models over their lifecycle. The SADL grammar is implemented in Xtext [40], which also provides the features of a modern IDE including semantic coloring, hyperlinking, outlining, and content assistance. The SADL grammar falls into two sections: it supports declaring the ontological concepts of a domain (classes, properties, individuals, property restrictions, and imports), and expressions that may appear in queries, rules, and tests.

As an illustration, consider CAPEC-28 which is Fuzzing. The NIST controls for it are SI-10 (Information Input Validation) and SI-10-5 (Restrict Inputs to Trusted Sources and Approved Formats). If CAPEC-28 impacting integrity ("CAPEC-28I" in rule below) is applicable, the mitigation ('applicableCM' in rule below) is InputValidation. CAPEC-28 is applicable to a component if the component involves software, is inside the trusted boundary and the data received is untrusted. This is captured as a rule in SADL shown in Figure 17.

```
Rule Vul-CAPEC-28-1
if oneOf(componentType of a Subsystem, Software, SwHwHybrid,
SwHumanHybrid,Hybrid)
  and insideTrustedBoundary of a Subsystem is true
  and dataReceivedIsUntrusted of the Subsystem is true
then applicableCM of the Subsystem is CAPEC-28I-InputValidation.
```

Figure 17. A STEM rule to identify components that are susceptible to CAPEC-28 input validation.

Attack-Defense Tree Analysis of System Architectures (SOTERIA++)

Attack trees are similar in nature to fault trees—they use the same logic gates and they both propagate events (either attack steps or component failures) from leaf nodes up to the root node. It is a disciplined and formal way of capturing attacks in a tree structure. The root node is the attack goal and the details on how to achieve that goal are the leaf nodes. Analysis of attack trees produces a logical, hierarchical, and graphic decomposition of attack paths and the conditions necessary for a threat to be realized. While they provide a formal and methodical way of thinking and describing threats, building attack trees is quite manual. Our philosophy, once again, is to provide a compositional, model-based framework to shift the engineer's focus away from generating and maintaining these artifacts. SOTERIA++ generates attack trees automatically, more specifically, attack-defense tree which we will describe in more details.

Attack trees are one arsenal of many, but it's one that's used widely in industry and is accepted by agencies and certification authorities. Other than it being widely accepted, it

has many foundational aspects that make it a decent security artifact. Attack trees were first conceived from the information security industry in 1999, accredited to Bruce Schneier [41], but are generally applicable and are not restricted to analyzing information system. Attack trees are focused on defining an attack and refining it from the attacker's point of view. They integrate well with the ancestor, the fault tree, which is focused on an undesired system failure as its top-level event. Marrying the two provides a system design tool that analyzes both safety and security. There are several recent works to extend attack trees. One is a multi-parameter attack tree to account for multiple interdependent parameters in each node [42]. Another is the inclusion of countermeasures within the attack tree, called an attack-defense tree [43,44]. According to the authors, an attack-defense tree is a better representation of a system over attack trees because the latter only captures attack scenarios and does not model the interaction between attacks and defenses that could be put in place to guard against the attacks. More importantly, system security is constantly evolving—as better control measures are put in place, more sophisticated attacks are implemented. Therefore, modeling only attacks without considering the defenses in place is very limiting. Guided by the formalisms introduced in [43,44], we extended their concepts to include guidelines and considerations from DO-326A and DO-356A so that the terminology used in the tree is relevant. Furthermore, we defined precisely the qualitative and quantitative aspects of the attack-defense tree, because just as fault trees are rooted to the theory of probability, we wanted our attack-defense trees to be grounded in theory from mathematics. The theoretical foundations are covered in [45], therefore a short summary is provided in the following paragraphs.

One of the terminologies in DO-356 is "likelihood of successful attack". The top node of an attack-defense tree represents an attacker's goal. The quantitative measure for fault trees is probability; the quantitative measure we defined for the attack-defense tree is likelihood of successful attack. Likelihood is not a probability and does not follow a probability distribution. Ref. [46] distinguishes the terms in this way: "Probability attaches to possible results; likelihood attaches to hypotheses . . . The results to which probabilities attach are mutually exclusive and exhaustive; the hypotheses to which likelihoods attach are often neither". Another interesting statement from this article is that, "Because we generally do not entertain the full set of alternative hypotheses . . . , the likelihoods that we attach our hypotheses do not have any meaning in and of themselves; only the relative likelihoods—that is, the ratio of two likelihoods—have meaning." VERDICT reports to the user likelihoods for each cyber requirement that can be compared across the system architecture being analyzed. The user should not attempt to compare the safety requirement probabilities against the cyber requirement likelihoods.

An attack-defense tree is made up of two types of nodes: attack nodes and defense nodes. The attack nodes come from STEM: based on component properties annotated on the AADL system components and connections (see Figure 3), STEM outputs CAPEC attacks. These attacks are parallel to the undesired events in safety analysis. In safety, undesired events such as loss of availability or integrity of a component are identified by an engineer either through lab tests or manufacturer specifications and specified in the VERDICT annex as Event (see Figure 9). In security, the attacks are identified by STEM and are always given a value of 1 for likelihood of success of attack. Assigning a number to the level of attack is quite difficult and will only hold true for a short period of time. According to Javaid et al. [47], "more emphasis should be put on countermeasures for threats". Therefore, in SOTERIA++ we assume a worst-case number for attacks and focus on assigning a score for the defenses.

STEM also returns suggested defenses for each attack. The user specifies which of these suggested defenses have been implemented in the actual system by annotating the AADL component with the right properties (see Figure 3, "Cyber Defense and Mitigation DAL"). The VERDICT annex allows the user to specify the Design Assurance Level (DAL) which indicates the level of rigor applied to implementing the mitigation on the system. DAL is how the likelihood of successful attack gets lowered.

An illustration of the security analysis result of the Delivery Drone System is presented in Figure 2. The security requirement presented in Figure 5 says, "*The drone shall be resilient to loss of ability to deliver a package to the appropriate consumer location.*" More formally, this is a loss of integrity concern on the Delivery Drone System, as specified by the field "cia=I". Furthermore, any of the following system outputs can contribute to this loss: loss of integrity of *actuation_out*, loss of availability of *actuation_out*, loss of integrity of *delivery_status*, or loss of availability of *delivery_status*. Finally, the severity of not meeting this requirement is labeled as "*Hazardous*", which equates to 1e-7. This means that identified attacks that impact *CyberReq01* must be designed with enough rigor to lower the likelihood of successful attack to 1e-7 or less. Having modeled compositionally the security relations of each component of the Delivery Drone (such as the one illustrated in Figure 8 for the *DeliveryItemMechanism*) and having modeled the component connections of the Delivery Drone System, the attack-defense tree snippet for *CyberReq01* is illustrated in Figure 18.

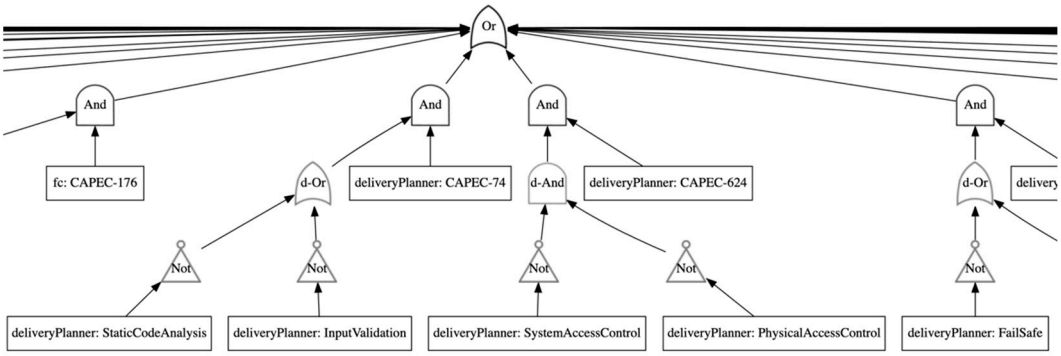


Figure 18. The attack-defense tree snippet for CyberReq01.

Synthesis of Optimal Defenses for Cyber-Vulnerabilities of System Architectures

Model-based Architecture Synthesis is to synthesize a minimal set of defenses with Design Assurance Level (DAL) while minimizing their implementation costs and satisfying all the cyber requirements. There are several operating modes for the synthesis tool based on the input. Synthesis can ignore the existing implemented defenses and yield a globally optimal defense solution, or it can find a locally optimal defense solution where it uses the existing implemented defenses. Synthesis with implemented defenses has two modes depending on whether the existing implemented defenses satisfy all the cyber requirements or not. The idea of synthesis is to transform attack-defense tree to only defense tree, where each defense node is associated with a user-specified cost. Then the defense tree with costs is further encoded into logical formulas in MaxSMT.

- Cost modeling for defenses

To run synthesis, the user needs to provide an implementation cost model for all possible combinations of component, defense property, and DAL. The cost model may be specified under linear scaling mode. Designers need to provide a scaling factor, and then the cost for a <component-property-DAL> triple will be the DAL multiplied by the scaling factor, thus is monotonic increasing with respect to DAL (i.e., higher DAL demands higher costs in general). The cost model is specified in an XML file in compliance with XML syntax and must be named "costModel.xml" placed at the top-level of the AADL project repository. In addition, the interpreter of the cost modeling abides by the following rules:

a. By default (no costs specified), a scaling factor of 1 is used for all <component-property-DAL> triples.
b. Unless a scaling factor for the generic cases is specified as shown in item #2 below, a scaling factor of 1 will be used for all unspecified costs.

c. More specific costs take precedence over less specific costs. For instance, costs specified in item #4 below take precedence over costs specified in item #2 and #3; and costs specified in item #3 take precedence over costs specified in item #2.

We will use the delivery drone example to demonstrate the XML syntax to encode different scenarios under linearly scaling cost mode.

1. Default mode (a scaling factor of 1 will be used for all <component-defense-DAL> triples): `<?xml version=''1.0'' encoding=''UTF-8'' standalone=''no''?> <cost-model> </cost-model>`
2. A scaling factor of 2.0 for all <component-defense-DAL> triples `<cost>2.0</cost>`
3. A scaling factor of 3.0 for all defense properties of the *actuation* subcomponent of *DeliveryDroneSystem.Impl*. (Note that the triple colon ":::" is used to concatenate the component implementation with the subcomponent instance. Also, the XML field keyword for connections is also component.) `<cost component=''DeliveryDroneSystem.Impl:::actuation''>3.0</cost>`
4. A scaling factor of 4.0 for the "*antiJaming*" defense property of the *camera* subcomponent of *DeliveryDroneSystem.Impl*. `<costcomponent=''DeliveryDroneSystem.Impl:::camera'' defense=''antiJamming''>4.0</cost>`

- Globally optimal defense solution

Under this mode, synthesis will synthesize a globally optimal defense solution that satisfy all cyber-requirements without considering existing implemented defenses. The solution is globally optimal with respect to the set of NIST-800-53 controls supported by VERDICT. It is a minimal (not necessarily unique) set of entity-defense-DAL triples. The synthesis problem is reduced to MaxSMT to be solved with Z3 solver [48]. Satisfiability Modulo Theories (SMT) task is to check the satisfiability of first-order formulas containing constraints from various theories such as arithmetic, arrays, and strings. MaxSMT extending SMTs task is to solve optimization problems over SMT formulas. We give a high-level overview of the encoding algorithm. To obtain a globally optimal defense solution, all applicable defenses are generated by STEM (every attack has a defense in STEM), and costs are provided by user. We create a SMT variable for each component-defense pair (p, d) as V_{p_d}. Note, an individual component-defense pair may occur multiple times in the attack-defense tree, but each occurrence is represented by the same variable in SMT. We define two functions f_{AD} and f_D. to transform an attack-defense tree to SMT encodings inductively (prefix notation is used), where $AND/OR_{AD}(x_1, ..., x_k)$ and $AND/OR_D(y_1, ..., y_k)$ denote attack nodes and defense nodes respectively; a, t, C(p, d, l) denote a CAPEC attack, a defense tree and the cost corresponding to the component p, defense d, and DAL l respectively. DAL l is chosen based on the severity in cyber requirement: 0 to No Effect; 3 to Minor; 5 to Major; 7 to Hazardous; and 9 to Catastrophic.

- $f_{AD}(AND_{AD}(x_1, ..., x_k))$ => (or $f_{AD}(x_1)$... $f_{AD}(x_k)$)
- $f_{AD}(OR_{AD}(x_1, ..., x_k))$ => (and $f_{AD}(x_1)$ ... $f_{AD}(x_k)$)
- $f_{AD}(ATTACK(a, p, t, l))$ => $f_D(p, t, l)$
- $f_D(p, AND_D(y_1, ..., y_k), l)$ => (and $f_D(p, y_1, l)$... $f_D(p, x_k, l)$)
- $f_D(p, OR_D(y_1, ..., y_k), l)$ => (or $f_D(p, y_1, l)$... $f_D(p, y_k, l)$)
- $f_D(p, DEFENSE(d), l)$ => (>= v_{p_d} C(p, d, l))

The objective function to be minimized by MaxSMT solvers is the sum of all implementation costs. It is important to note that when encoding the attack-defense nodes, we flip the AND and OR nodes. This transposition follows from the definitions, but the intuitive explanation is that attack-defense nodes are effectively opposites of their defense node counterparts because the attack-defense nodes are from the perspective of the attacker and the defense nodes are from the perspective of the defender. The model returned by the solver is constructed for each component p and defense d by using the inverse function of $C^{-1}(c) = \max\{l \mid C(p, d, l) = c\}$. Essentially, the inverse selects the maximum DAL with the given cost.

A globally optimal defense solution example for the delivery drone example returned by the synthesis is shown in Figure 19. There are five columns on the result panel: (1) Designer Action shows what the designer has to do, (2) Component/Connection shows to which component the designer has to take the action, (3) Defense Property shows the defense property that the designer has to act on, (4) Target DAL shows the DAL level that the designer has to implement the defense property, and (5) Target Cost shows the costs to implement the defense property.

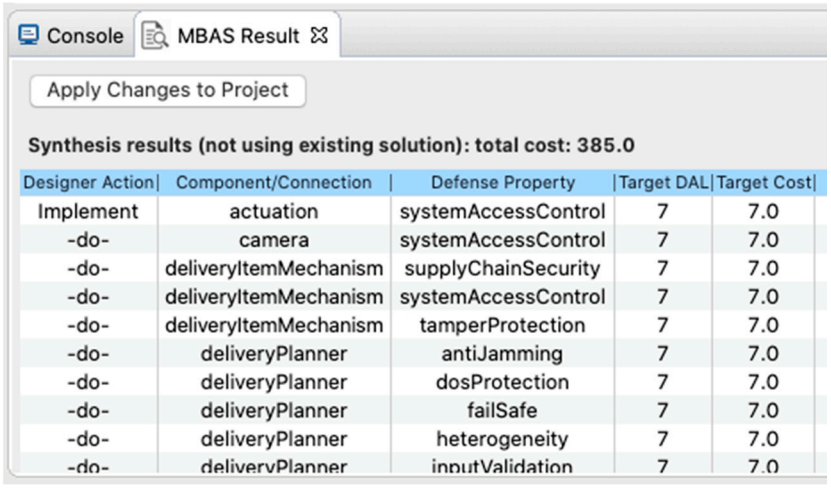


Figure 19. A globally optimal defense solution (incomplete picture) for the delivery drone example returned by the synthesis.

- Locally optimal defense solution

Under this mode, synthesis will synthesize a locally optimal solution using existing implemented defenses based on whether the implemented defenses satisfy the cyber requirements or not. In the case that implemented defenses do not satisfy the cyber requirements, synthesis selects a minimal set of component-defense-DAL triples to implement or upgrade in order to satisfy all cyber requirements. An example of such synthesis solution is shown in Figure 20. This mode considers the already-implemented defenses to be free. The output of the tool still specifies the cost of the already-implemented defenses. The implemented defenses are free for the purposes of minimizing total cost. This solution is minimal among the solutions that can be obtained without removing or downgrading any defenses. Note that there may be already implemented defenses that are unnecessary, which if removed (as recommended by the other mode below) will yield a solution with smaller cost. In this case, additional SMT constraint is introduced to encode the cost for the implemented DAL as lower bound for each component and defense. Thus, (>= $v_{p'\_d'}$ C(p', d', l')), where l' is the implemented DAL for the component-defense pair (p', d').

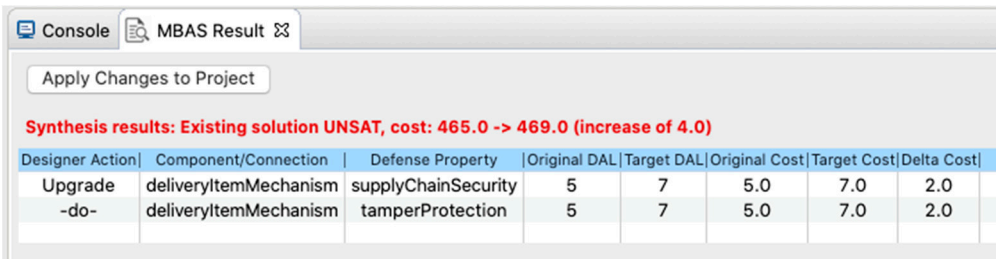
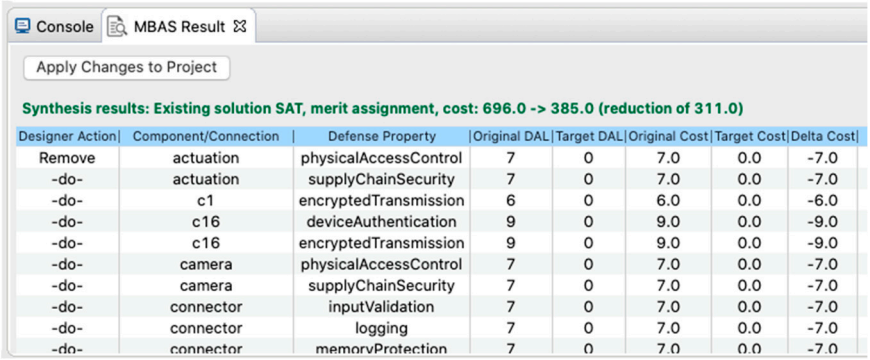


Figure 20. An example of locally optimal defense solution for the case that implemented defenses do not satisfy cyber requirements.

Conversely, if the implemented defenses already satisfy the cyber requirements, synthesis performs additional cost reductions, which is to select a maximal set of entity-defense-DAL triples to remove or downgrade while still satisfying the cyber requirements. In essence, we no longer treat the already-implemented defenses as free. This solution is minimal among solutions that can be obtained without implementing or upgrading any defenses. The resulting solution is locally minimal because achieving a more minimal solution, if one exists, requires removing or downgrading some defenses and implementing or upgrading others. In this case, additional SMT constraint is introduced to encode the cost for the implemented DAL as upper bound for each component and defense. Thus, (<= $v_{p'\_d'}$ C(p', d', l')), where l' is the implemented DAL for the component-defense pair (p', d'). An example of the synthesis solution for this case is shown in Figure 21.


Console MBAS Result

Apply Changes to Project

Synthesis results: Existing solution SAT, merit assignment, cost: 696.0 -> 385.0 (reduction of 311.0)

Designer Action	Component/Connection	Defense Property	Original DAL	Target DAL	Original Cost	Target Cost	Delta Cost
Remove	actuation	physicalAccessControl	7	0	7.0	0.0	-7.0
-do-	actuation	supplyChainSecurity	7	0	7.0	0.0	-7.0
-do-	c1	encryptedTransmission	6	0	6.0	0.0	-6.0
-do-	c16	deviceAuthentication	9	0	9.0	0.0	-9.0
-do-	c16	encryptedTransmission	9	0	9.0	0.0	-9.0
-do-	camera	physicalAccessControl	7	0	7.0	0.0	-7.0
-do-	camera	supplyChainSecurity	7	0	7.0	0.0	-7.0
-do-	connector	inputValidation	7	0	7.0	0.0	-7.0
-do-	connector	logging	7	0	7.0	0.0	-7.0
-do-	connector	memoryProtection	7	0	7.0	0.0	-7.0

Figure 21. An example of locally optimal defense solution for the case that implemented defenses satisfy cyber requirements. For the sake of space, only partial content is shown.

5.4. Assurance Case Fragments Generation

An assurance case is an argument-based documentation that can provide explicit assurance that a system satisfies certain desired safety, security, or reliability properties. Assurance cases have been widely used for system certification in industry as they are easy to understand by domain experts and certifiers and they provide structured and irrefutable arguments about system requirements. We have, therefore, incorporated in VERDICT toolchain the functionality of generating both safety and security assurance case fragments automatically to provide system designers with an option to quickly assemble the assurance evidence in an industry-accepted format for certification.

Multiple standards exist for assurance case representation such as CAE (Claim, Argument and Evidence) [49], GSN (Goal Structuring Notation) [50,51], and SACM (Structured Assurance Case Metamodel) [52]. However, all approaches involve three basic elements: claims, arguments, and evidences, which justify how the evidences can satisfy the claims. In VERDICT, we use the GSN notation for representing assurance case fragments because it provides a graphical representation of the elements, which can be more easily analyzed by human designers and certifiers than non-graphical representations. The GSN standard contains four principal elements that are connected to create a structured argument in the form of a graph as shown in Figure 22. They are:

- Goals represent the requirements a system is expected to satisfy.
- Solutions are lowest-level evidence that may be used to claim that a goal has been satisfied.
- Strategies state how goals are satisfied by sub-goals or solutions.
- Contexts contain contextual information and can be used with goals, solutions, or strategies.

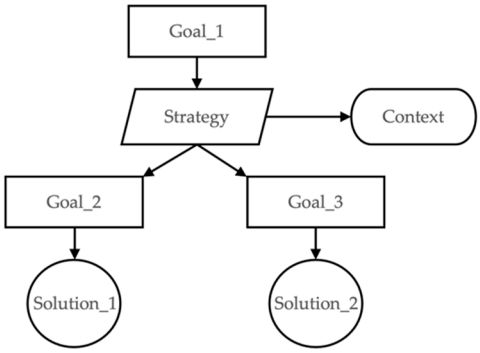


Figure 22. An example of GSN graph.

5.4.1. GSN Assurance Case Fragments in VERDICT

A meaningful and complete GSN fragment should include the four components mentioned above to form a comprehensible and structured argument about the requirements of a system. In the context of the VERDICT toolchain, a system can have three types of requirements—mission requirements, cyber requirements, and safety requirements. The information present in the corresponding constructs in the verdict annex for these requirements can be used to populate the goals, strategies and contexts of a GSN fragment. Currently, the assurance case tool only supports evidence generated after MBAA which are available in the form of XML outputs from SOTERIA++. We explain below how this information is used to generate the various components of the VERDICT GSN fragments.

- Goals are requirements specified in the VERDICT annex. The highest-level goals are the mission requirements, which are supported by cyber and safety requirements. The goal statement is the statement declared in the description construct of a requirement.
- Strategies are used to connect each goal with its sub-goals or to the SOTERIA++ solutions that prove or disprove them. For mission-level goals, the strategies argue correctness by validity of sub-goals. For cyber-level and safety-level goals, the strategies argue correctness by SOTERIA++ analysis of attack-defense trees and fault-trees respectively.
- Contexts are provided in VERDICT as follows:
    1. For mission-level goals, a context is what the requirement is concerned for the system.
    2. For strategies of mission-level goals, a context describes the cyber and safety requirements specified in the *reqs* construct of the mission requirements.
    3. For cyber-level and safety-level goals, contexts are provided for the ports specified in the *condition* construct of the requirements.
    4. For strategies of cyber-level and safety-level goals, a context provides reference to the information present in the *condition*, *severity*, and *targetprobability* constructs of the requirements and another context provides reference to the VERDICT properties associated with the system model.
- Solutions: two types of evidence are provided by VERDICT and are used as solutions in the VERDICT GSN:
    1. Likelihood of minimal cut-sets for cyber requirements generated by SOTERIA++.
    2. Probability of minimal cut-sets for safety requirements computed by SOTERIA++.
- Extensions to the GSN standard

When flaws in a system model cause one or more requirements to fail during the analysis, the solutions that are generated from SOTERIA++ cannot support the goals that they are expected to support. Under such circumstances, complete assurance case fragments cannot be generated for the higher-level goals that are supported by lower-level goals that have failed. However, even the information about failed goals can be presented in

a graphical manner that is beneficial to system designers. We have extended the VERDICT GSN by providing colored indications about the success or failure of goals, strategies, and solutions in a GSN. The rules that dictate the color of a node are given below:

1. If the SOTERIA++ output for a cyber requirement has computed likelihood > acceptable likelihood, then the solution has failed and is colored red. Otherwise it is colored green.
2. If the SOTERIA++ output for a safety requirement has computed probability > acceptable probability, then the solution has failed and is colored red. Otherwise it is colored green.
3. If a strategy is supported by a solution, it bears the same color as the solution.
4. If a strategy is supported by one or more goals, it is colored green if and only if all supporting goals are green. Otherwise it is colored red.
5. A goal bears the same color as the supporting strategy.

We refer to a GSN graph in which all goals, strategies, and solutions are green as a complete assurance case fragment. If a GSN graph has at least one red node, it is an incomplete assurance case fragment, but can still be useful to designers as it provides a clear visual indication of which goals and sub-goals have failed, allowing the designers to easily isolate, locate, and mitigate flaws in a model.

The GSN notation allows additional supporting elements for incorporating information in assurance case fragments. We have also extended the VERDICT annex with two optional String constructs called justification and assumption. In a GSN graph, justifications and assumptions are represented by oval nodes.

5.4.2. An Illustration of Assurance Case Fragments Generation in VERDICT

The ACFG of VERDICT tool-suite generates three types of artifacts for each GSN fragment (1) a dot file; (2) an SVG file for the GSN graphs and (3) an XML file with the GSN data embedded. The dot file is intended for rendering SVG files, and the SVG file displays GSN assurance case fragments visually. The XML file is used for data exchange with other assurance case tools. An SVG graph displaying the assurance case fragments for the mission requirement MReq01 of the delivery drone example is shown in Figure 23. A GSN example generated by VERDICT for the mission requirement *MReq01*. The example shows three branches with one failing branch colored red and two succeeding branches colored green.

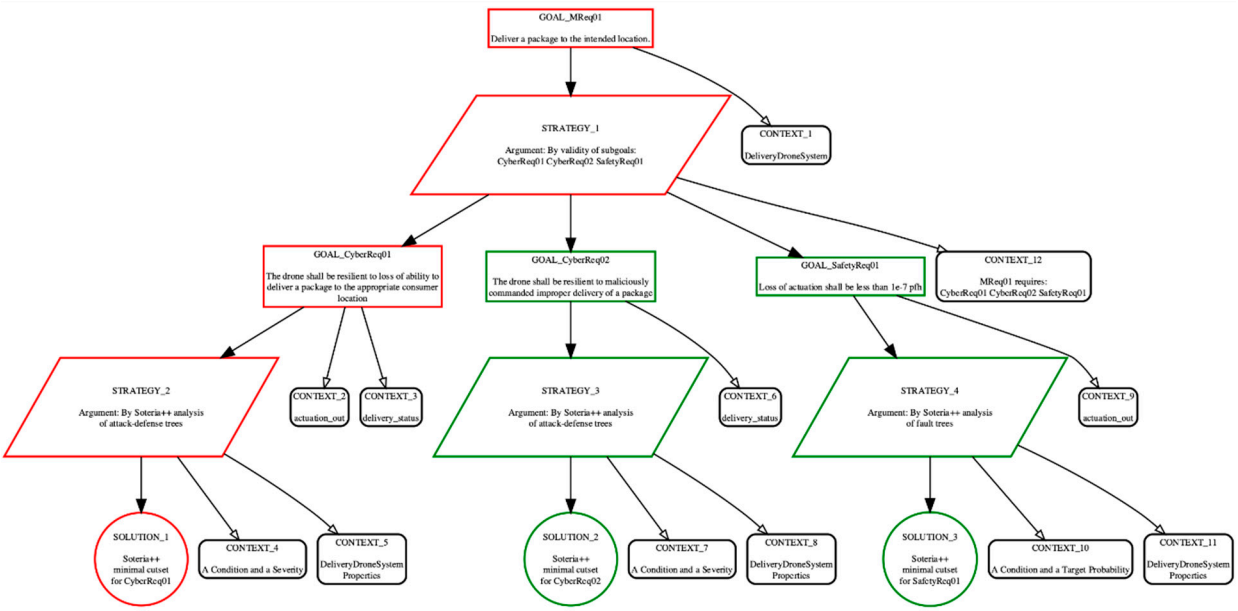


Figure 23. A GSN example generated by VERDICT for the mission requirement *MReq01*.

In addition, ACFG could generate fine-grain views of assurance case fragments with user-settings. Examples of GSN security assurance case fragments for cyber requirement *CyberReq02* (top) and *deliveryItemMechanism* component (bottom) are shown in Figure 24. A fine-grain view of assurance case fragments for cyber requirement *CyberReq02* and for the *deliveryItemMechanism* component. Each solution node in the graph is clickable, which can redirect to the evidence file.

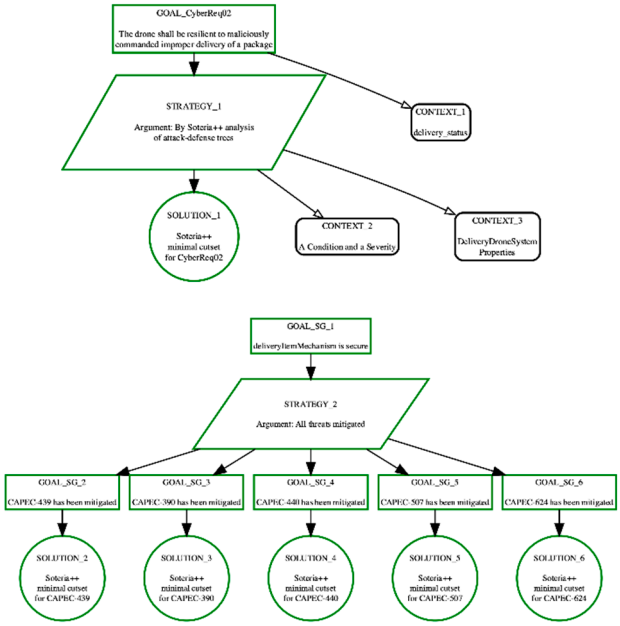


Figure 24. A fine-grain view of assurance case fragments for cyber requirement *CyberReq02* (top) and for the *deliveryItemMechanism* component (bottom).

5.5. Cyber Resiliency Verifier (CRV)

In our context, we say a system is cyber-resilient with respect to a given set of desired functional properties and threat models only if the system can ensure the satisfaction of these properties even when the system is susceptible to attacks considered in the threat models. The cyber-resiliency verifier (CRV) enhances the VERDICT tool's capability of performing architecture-level security and safety analysis with the capability of formally reasoning about a system design's cyber-resiliency. More precisely, given a system design and a list of functional properties (expressed in some formal logic) that the system architect expects the design to satisfy, CRV checks to see whether the design is *resilient to integrity attacks*, that is, the stipulated functional properties can be guaranteed even when some components and channels of the system are vulnerable to integrity attacks (e.g., logic bomb, network spoofing, remote code execution, control-flow integrity violation). In our context, a system design consists of not only the architecture of the system (i.e., components and their interconnections) but also the behavioral information of each system component. Once a system design along with formal properties have been added to the system design model (or, just system model) under analysis, as discussed before, CRV allows one to analyze the system in both a benign case as well as under a library of different threat models. Currently, the library currently encompasses eight threats: logic bomb, insider threat, location spoofing, network injection, hardware trojans, outside user threat, remote code injection, and software virus/malware/worm/trojan, but can be extended as needed. In case CRV can identify an execution under which one of the desired functional properties can be violated, either in the benign case or in the adversarial case, as an evidence it generates a counterexample, which is a sequence of execution steps of the system that demonstrates the violation. In addition, it generates some other diagnostic information in the forms of *blame assignment* (when a counterexample is discovered) and *merit assignment* (when no counterexamples exist). These additional two pieces of diagnostic information

can be used by the designer to refine the design to avoid violating a desired property and hence making the system more cyber-resilient.

Given a set of threat models, the cyber-resiliency analysis performed by CRV on the original design model can be reduced to a model checking problem of an instrumented model where the instrumented model is an enhancement of the original one while considering integrity attacks according to the input threat models. Our automatic instrumentation process is enabled by the novel notion of *attack effects* instead of concrete attacks. By capturing attack effects that influence a system component's/channel's integrity, one can abstractly group together multiple attacks. In addition, one can simulate attack effects by carefully placing non-determinisms. As an example, let us assume a system design contains two components A and B such that A takes its input from the environment whereas A's outputs are fed into B's inputs, and B's outputs are considered to be system outputs. Now, if we consider that A is vulnerable to control-flow integrity attack (e.g., buffer overflow, ROP), then component A can behave arbitrarily according to the adversary's choice. To capture such integrity attacks on A's behavior, in our instrumented model, we can just introduce non-determinism in the logical channel that is carrying information from A's outputs to B's inputs. In short, this can be viewed as simulating a lossy channel that does not always deliver A's outputs to B's inputs faithfully (i.e., the lossy channel can modify A's output arbitrarily). As a side-note, we introduce an enable switch for each of the non-determinisms we place in the form of a Boolean variable. When the Boolean variable is set to *true* the channel acts in lossy channel whereas *false* value of the Boolean enable switch indicates the channel carries information faithfully. These switches are considered to be as a model parameter (i.e., symbolic value) when considering blame assignment analysis and will be made clear later.

We essentially use non-determinism to consider all possible attack strategies modifying the behavior of A (e.g., after the ROP attack, we do not know what code the attacker would run in place of A's original code) and in turn using the model checker as an attacker to find plausible attacker strategies to violate the desired functional properties. In a typical model checking context, an attacker can only control the system inputs whereas we extend this influence to internal communication between components and claim that it is sufficient to simulate integrity attacks on components and channels.

Advantage of CRV's Analysis: Considering the design model enables CRV to more precisely identify or rule out security vulnerabilities of a system beyond what was just analyzed at the architectural level. For example, suppose a system uses a GPS component for navigation. By only inspecting the architecture of a system, the best we could do at the architectural level is to identify that the system is possibly vulnerable to a GPS spoofing attack. Even if the system uses a robust GPS sensor to resist spoofing attacks, due to the lack of behavioral details in the architecture we would not be able to rule out such attacks. Thanks to its access to the design model, however, CRV is able to rule out such false positives. Furthermore, it may identify vulnerabilities missed by inspecting only the system architecture. For instance, suppose a system includes other location sensors (e.g., a LIDAR) together with the GPS sensor and uses a majority voting scheme to rule out spoofing attacks under the assumption that an attacker cannot simultaneously compromise/attack all the positioning sensors on-board. Such a system will be deemed resistant against spoofing attacks by architecture analysis tools. However, by the way this security-enhancing mechanism is designed, it can have logical errors resulting in spoofing attacks. For instance, the majority voting scheme design in the above scenario could have a vulnerability that, under certain conditions, makes it adjudicate the spoofed location value to be the correct value. CRV can identify such a situation through a formal analysis of the components' behavior.

Another way that CRV adds to the analysis is in the threat model it uses for its analysis. More precisely, architectural analysis tools only consider previously documented attacks as part of their threat model. In contrast, CRV groups possible attacks based on their effects on the system and collectively reasons about them considering an abstract threat effect model.

For example, buffer/heap overflow, SQL injection, and return-oriented-programming (ROP) attacks, in the worst case, all provide an attacker with the ability to run malicious code in a system without having prior proper privileges. CRV will thus consider "running malicious code" as a possible effect instead of considering all known attacks that achieve this effect. Such an approach has a number of clear benefits. First, the number of effects one must consider for covering a wide variety of attacks is substantially smaller than the number of concrete attacks. Second, this approach can capture also unknown or even future attacks if these attacks have an effect already captured in the threat effect model.

Detailed CRV Workflow: CRV is meant to be used in the system design phase. Its workflow is depicted in Figure 25. The system architect/designer is responsible for providing the system design model (refer to 1) as input to CRV. The designer is also responsible for choosing an appropriate threat effect model (refer to 2) from a library of such models under which to carry out the necessary formal resiliency analysis. The CRV workflow intentionally separates the system's design model from the adversarial threat effect models. This decoupling enables the system designer to develop the design model without having to include aspects of the system's security properties. Given the system design model, the chosen threat effect model, as well as a number of desired cyber-resiliency properties for the system, the threat model instrumentor (refer to 3) first automatically composes the two model to obtain a threat-instrumented model (refer to 4). Then, the satisfaction of the cyber-resiliency properties (refer to 5) on the threat-instrumented model is checked by the reasoning engine powered by the Kind 2 model checker (refer to 6) [53]. Cyber-resiliency properties are provided by the designer and expressed in past-time linear temporal logic. The analysis artifacts produced as feedback (refer to 7) by CRV include the counterexample, the localized diagnosis information to trace possibly vulnerable components in the system (blame assignment) and identifying components who may have played a critical role in verifying the cyber-resiliency requirement even in an adversarial environment (merit assignment).

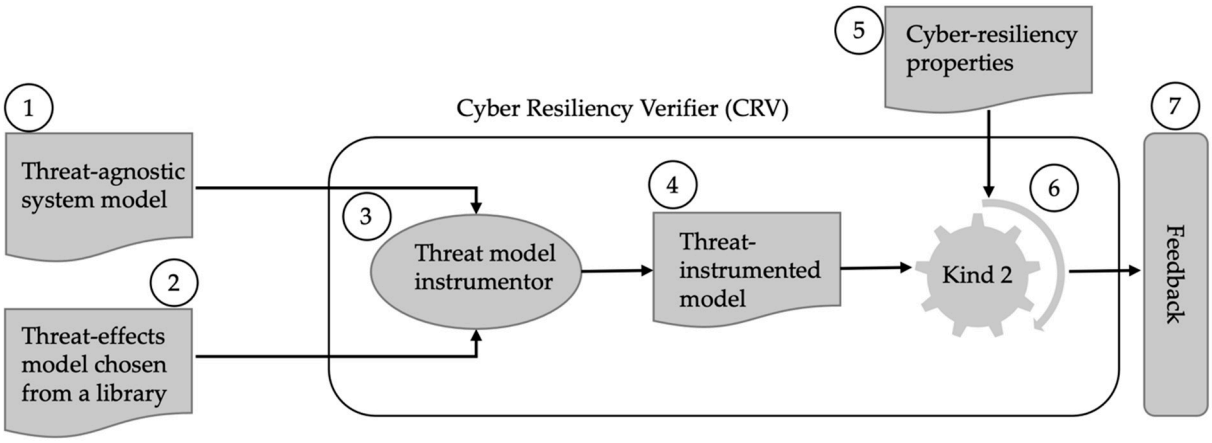


Figure 25. Detailed CRV workflow adapted from [1]. The threat agnostic system model (1) and threat-effects model (2) are fed to the threat model instrumentor (3), which generates a threat instrumented model (4). Cyber-resiliency properties (5) together with the instrumented model are then passed to the Kind 2 model checker (6), which yields feedback (7).

We envision one to carry out the analysis of their system design with respect to the formal property in the benign case before moving on to the adversarial case. This recommendation is to ensure that the system can verifiably maintain the required functional formal properties when there are no adversarial interferences. If this analysis step were to be skipped, and one directly starts the analysis with respect to some threat models in the CRV library, then if CRV were to generate a counterexample, it is difficult for the user to pinpoint whether the violation is due to adversarial interference or a design flaw. Once a user has verified their design in the benign case, when they move on to the adversarial

case and observes a counterexample, they can be sure that the violation is due to the adversarial interference.

5.5.1. Blame Assignment

When the instrumented model is fed into the model checker during CRV workflow, for each formal property, the model checker can come back with one of the following three possible answers: *Safe* (signifying the model checker was able to prove that the model satisfies the property); *Unsafe* (signifying the model does not satisfy the property); *Unknown* (signifying the model checker timed out and could not prove/disprove the property). For each property determined to be *unsafe*, in addition to the attack trace, CRV can also generate information regarding misbehaving components that may have contributed to the violation. We call this functionality blame assignment. CRV supports a *locally optimized* and a *globally optimized* forms of blame assignment both of which are achieved by posing a series of queries to the backend model checker.

Technically, the blame assignment problem is an optimization problem that requires minimizing the number of enabled switches (i.e., the number of vulnerable components/channels) required to obtain a counterexample. This can be viewed as identifying a minimum cut set of enabled switches that is sufficient for demonstrating a violation of the property in question.

Suppose during instrumentation we introduce non-determinisms in n channels, each of which is controlled by a unique enable switch (i.e., a Boolean variable), and after analysis the model checker discovered a counterexample containing l steps of the system execution. The locally optimized form of blame assignment checks to see whether it is possible to generate a l-step counterexample using a smaller number of enable switches turned on compared to n. Note that, the counterexample observed during the minimization step of locally optimized blame assignment may not be identical to the one observed during initial model checking; it just needs to be of the same length l. The globally optimized form of blame assignment is similar to the locally optimized one with one exception: there is no restriction of the counterexample length being equal to the original one obtained during model checking. The insight is that it may be possible to generate a counterexample with a smaller number of enable switches turned on (i.e., vulnerable components/channels) when there are no restrictions that the obtained counterexample during this analysis is of the same length as the one obtained during original model checking.

The algorithm: We now present our algorithm for finding the minimum cut set, first in the global optimization setting and then in the local optimization one. In the original model checking problem, all switch parameters are all set to *true*. For blame assignment, however, all switch parameters are unset. Solving the blame assignment problem then corresponds to finding a truth assignment to these parameters that produces a violation of the property with the smallest number of parameters set to *true*. In other words, let S and O denote the total number of enabling switches and the total number of those that are turned on, respectively. We want to minimize the value of O while still violating the property. For this, we pose a series of model checking queries with the following two additional restrictions added to the original model checking problem:

- (R1) each switch should maintain the same value during the whole execution (that is, it cannot switch between on and off);
- (R2) O is bounded above by some constant C ($O \leq C$).

Initially, we choose $C = S - 1$ and keep decreasing the value of C as long as the model checker returns unsafe for the enhanced model with the two restrictions above. Once we find the smallest value of O, we identify from the attack trace the components whose enabling switches are turned on and return the corresponding compromised component in the blame assignment.

In the global optimization setting, the attack trace corresponding to the smallest blame assignment set can be longer than the original trace, the one found by the model checker before the optimization. If, on the other hand, the designer is interested in attack traces of

minimal length, it is possible to ask CRV for a locally optimized blame assignment. In this setting, we generate a bounded model checking problem [54] to find attack traces of the same length as the original trace, but we force a MaxSMT solver to minimize the number of parameters set to *true* by setting the negation of the Boolean switch parameters as *soft* constraints. This produces the desired effect if that trace is already of minimum length.

5.5.2. Merit Assignment

For merit assignment, which is relevant when the system satisfies the given cyber-resiliency properties even when under attack, CRV can provide traceability information between those properties and design elements such as assumptions, guarantees or other behavioral constraints. Specifically, CRV uses the new capability of Kind 2 for identifying minimal sets of design elements, known as Minimal Inductive Validity Cores (IVCs) [49], which are sufficient to prove the given set of cyber-resiliency properties. The functionality can be enabled as post-analysis in the CRV Settings panel. When CRV can prove that one or more properties are satisfied by the model, the user can view inductive validity core of the satisfied properties. It will show the components involved in the satisfaction of the property, and for each component, which assumptions and guarantees were used in the proof.

5.5.3. An Illustrative Example Using CRV

We now describe how a designer/user would interact with the CRV functionality of VERDICT on the delivery drone system in three cases: benign case (no threats against the system), threat effect case (CRV threats against the system), and mitigation case (a mitigation plan is implemented in the system).

- Benign case

Once we have extended the system design with behavioral information as well as formal properties of interest, the first thing a user has to check is that the system design model satisfies the property in a benign scenario, that is, without considering the effects of any threat model. To do that, the user would need to make sure to no CRV threat models is selected and then invoke Cyber Resilience Verifier (CRV). This initiates the translation of the AADL model to a VERDICT internal data model and then finally to the input to CRV. CRV will find a scenario where property P7 is violated. Since we know that cannot be the case in the benign scenario, one can add the following assumption to the *DeliveryDroneSystem* specification (Figure 26):

```
assume "Delivery locations set through bus are never off-limits locations":
  not Agree_Nodes::InRestrictedArea(bus1.order.target_position);
```

Figure 26. Assumption on the input defined in AGREE annex.

In the case, since no threat is enabled, property P7 is proved valid as shown in Figure 27.

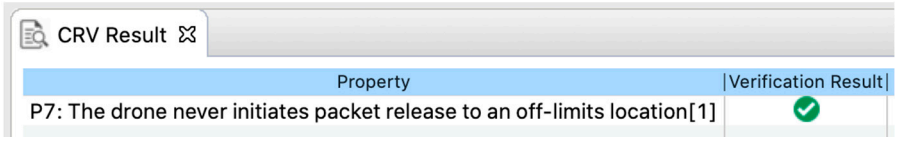
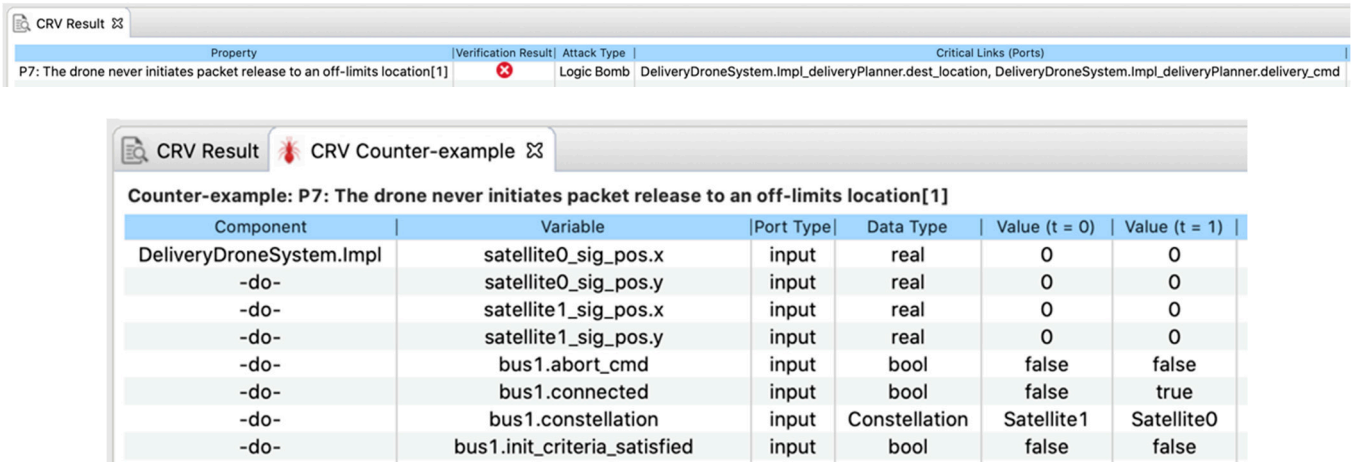


Figure 27. CRV result for property P7 in the benign case.

- Threat effect case

We choose to apply all CRV threats on the model. In this case, CRV will find a scenario where property P7 is violated. It is evident that CRV has detected a possible violation of property P7 due to a Logic Bomb that may be hidden in the logic of the *DeliveryPlanner*. Moreover, the blame assignment analysis identifies a minimal set of critical ports, namely

dest_location and *delivery_cmd*, that are sufficient for carrying out the Logic Bomb attack as shown in Figure 28. In addition, one can view counter-example to display trace that leads the system to the violation of the property. Examining the trace, one can observe that the property might be violated if the delivery location provided through the *bus1* is off-limits.



CRV Result

Property	Verification Result	Attack Type	Critical Links (Ports)
P7: The drone never initiates packet release to an off-limits location[1]	⊗	Logic Bomb	DeliveryDroneSystem.Impl_deliveryPlanner.dest_location, DeliveryDroneSystem.Impl_deliveryPlanner.delivery_cmd

CRV Result | CRV Counter-example

Counter-example: P7: The drone never initiates packet release to an off-limits location[1]

Component	Variable	Port Type	Data Type	Value (t = 0)	Value (t = 1)
DeliveryDroneSystem.Impl	satellite0_sig_pos.x	input	real	0	0
-do-	satellite0_sig_pos.y	input	real	0	0
-do-	satellite1_sig_pos.x	input	real	0	0
-do-	satellite1_sig_pos.y	input	real	0	0
-do-	bus1.abort_cmd	input	bool	false	false
-do-	bus1.connected	input	bool	false	true
-do-	bus1.constellation	input	Constellation	Satellite1	Satellite0
-do-	bus1.init_criteria_satisfied	input	bool	false	false

Figure 28. CRV Result (top) and counter-example (bottom) for property P7 under the threat effect case.

- Mitigation case

Once the vulnerable components and ports are identified, the user can consider multiple possibilities to address the root cause of the attack. Let us assume the designer decides to place a runtime monitor called "*PositioinRuntimeMonitor*" between the *DeliveryPlanner*, and the *DeliveryItemMechanism* that checks continuously whether the delivery location is off-limits, and raises a warning flag when it is the case and a command for releasing a package is issued. The AADL modeling of "*PositioinRuntimeMonitor*" is shown in Figure 29.

```
system PositionRuntimeMonitor
  features
    delivery_cmd_in : in data port Data_Types::PackageDeliveryCommand;
    delivery_cmd_out : out data port Data_Types::PackageDeliveryCommand;
    loc: in data port Data_Types::Position.impl;
    warning_flag: out data port Base_Types::Boolean;
    annex agree {**
     eq release_cmd: bool = (delivery_cmd_in = Agree_Constants::RELEASE_PACKAGE_CMD);
     guarantee "Delivery command is pass-through":
       delivery_cmd_out = delivery_cmd_in;
     guarantee "warning_flag value":
       warning_flag = (release_cmd and Agree_Nodes::InRestrictedArea(loc));
    **};
end PositionRuntimeMonitor;
```

Figure 29. AADL modeling of *PositionRunTimeMonitor* with behavior defined in AGREE annex.

With this runtime monitor in place, the *DeliveryItemMechanism* can be extended to use the warning flag to prevent the delivery of the package in a restricted area. For that, one will need to add a new input port *warning_flag* that reads the signal from the runtime monitor, and also add a new guarantee to the *DeliveryItemMechanism* to ignore the command when the signal is true. The definition is shown in Figure 30.

```
guarantee "If warning flag is set, delivery status is NOT_STARTED":
  true -> (warning_flag => delivery_status_out = Agree_Constants::NOT_STARTED_STATUS);
```

Figure 30. Guarantee statement in AGREE annex in *DeliveryItemMechanism* component to check the *warning_flag*.

A new analysis of the system confirms the applied fix is sufficient to rule out new Logic Bomb attacks as shown in Figure 31.

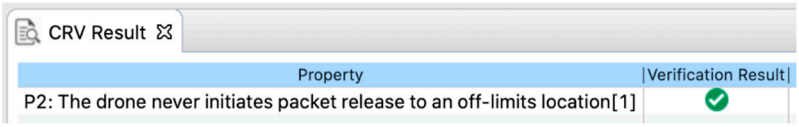


Figure 31. CRV analysis result after implementing behavioral defense.

Moreover, the *Merit Assignment* post-analysis can assure the user that the runtime monitor really plays an important role in the satisfaction of property P7 under the adversarial environment. The result is shown in Figure 32.

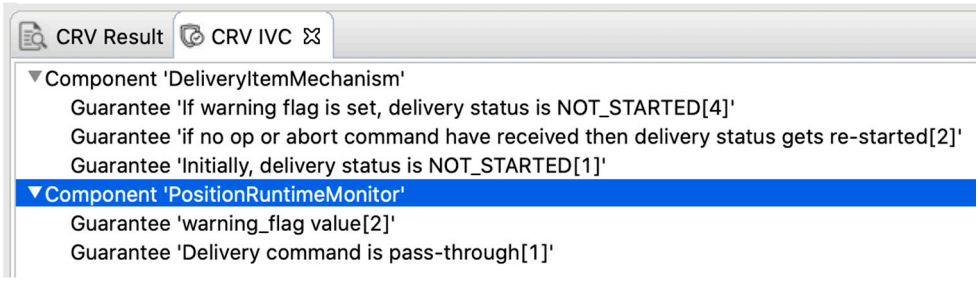


Figure 32. Merit Assignment analysis result after implementing behavioral defense.

In fact, thanks to the mitigation implemented, the assumption "*Delivery locations set through bus1 are never off-limits locations*", that the user added before during the analysis in the benign scenario, is not required anymore.

6. Conclusions

The DARPA Cyber Assured Systems Engineering (CASE) program goal was to develop design, analysis and verification tools to allow systems engineers to design cyber resiliency into complex embedded computing systems. The open-source VERDICT language and tool-suite developed on the CASE program were described in detail and demonstrated with illustrative examples. Links to the open-source code and example model are provided on GitHub where interested parties may download the code, training materials and working examples.

The VERDICT tool and domain specific language was developed as a plugin to the OSATE IDE. System engineers may develop and analyze their architectural and behavioral models in the OSATE environment where they create AADL models, then annotate them with VERDICT DSL, AGREE behavioral models and formal properties. The user may interactively call the Model-based Architecture Analysis and Synthesis function to identify CAPEC threats, NIST 800-53 defenses, generate resiliency metrics, generate faut trees, attack-defense trees and goal structuring notation fragments. The user may call the Cyber Resiliency Verifier function to append user selected threat effect models to the design then formally prove security properties with merit assignment and generate counterexamples along with blame assignment.

Future plans include proof of concept use of the tool on multiple product examples to evaluate the usability and benefit story. On the research side, the team intends to continue developing the analysis algorithms, connect to other high assurance tool chains and extend the front end to support SysML tools.

Author Contributions: Conceptualization, D.L., K.S., A.M., M.D., C.T. and O.C.; Formal analysis, B.M., K.S., A.M., W.S., D.P., M.D., C.T. and O.C.; Methodology, B.M., C.T. and O.C.; Project administration, M.D., C.T. and O.C.; Software, B.M., D.L., J.I., W.S., S.P., H.H.-Z., M.F.A. and M.Y.; Supervision, M.D., C.T. and O.C.; Writing—original draft, B.M., K.S., A.M., H.H.-Z., V.T.V. and O.C.; Writing—review & editing, B.M., O.C., D.L., A.M. and M.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was developed with funding partially from the Defense Advanced Research Projects Agency (DARPA). 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. DISTRIBUTION A. Approved for public release: Distribution Unlimited. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4006.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data used in this paper is publicly available at https://github.com/ge-high-assurance/VERDICT (accessed on 2 March 2021).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Table A1. Aligning CAPECs, VERDICT Cyber Defense Properties and NIST 800-53 Controls.

CAPECs with CIA info	Cyber Defense Properties	Conjunction of NIST Controls	
224:Fingerprinting [C]	auditMessageResponses	SI-11	
22:Exploiting Trust in Client [CIA]	deviceAuthentication	IA-3	IA-3-1
137:Parameter Injection [I]			
148:Content Spoofing [I]			
151:Identity Spoofing [CI]			
175:Code Inclusion [I]			
242:Code Injection [CIA]			
248:Command Injection [CIA]			
586:Object Injection [IA]			
594:Traffic Injection [IA]			
125:Flooding [A]	dosProtection	SC-5	SC-5-2
188:Reverse Engineering [C]	encryptedStorage	SC-28	
117:Interception [C]	encryptedTransmission	SC-8	
192:Protocol Analysis [C]			
148:Content Spoofing [I]	heterogeneity	SC-29	
151:Identity Spoofing [CI]			
28:Fuzzing [CIA]	inputValidation	SI-10	SI-10-5
148:Content Spoofing [I]	physicalAccessControl	PE-3	
188:Reverse Engineering [C]			
192:Protocol Analysis [C]			
440:Hardware Integrity Attack [I]			
507:Physical Theft [CIA]			
624:Fault Injection [CI]			
116:Excavation [C]	removeIdentifyingInformation	SI-15	
169:Footprinting [C]			
130:Excessive Allocation [A]	resourceAvailability	SC-6	
131:Resource Leak Exposure [A]			

Table A1. *Cont.*

CAPECs with CIA info	Cyber Defense Properties	Conjunction of NIST Controls				
21:Exploitation of Trusted Credentials [CI]	sessionAuthenticity	SC-23				
25:Forced Deadlock [A]	staticCodeAnalysis	SA-11-1				
26:Leveraging Race Conditions [CI]						
114:Authentication Abuse [CIA]						
115:Authentication Bypass [CIA]						
123:Buffer Manipulation [CIA]						
112:Brute Force [CIA]	strongCryptoAlgorithms	SC-13				
148:Content Spoofing [I]	systemAccessControl	PE-3	PE-3-1			
188:Reverse Engineering [C]						
192:Protocol Analysis [C]						
390:Bypassing Physical Security [CIA]						
440:Hardware Integrity Attack [I]						
507:Physical Theft [CIA]						
624:Fault Injection [CI]						
188:Reverse Engineering [C]	tamperProtection	SA-18-1				
440:Hardware Integrity Attack [I]						
507:Physical Theft [C]	zeroize	MP-6-8				
607:Obstruction [A]	antiJamming	SC-40	SC-40-1			
	failSafe	SI-17				
137:Parameter Injection [I]	inputValidation	SI-10	SI-10-5			
175:Code Inclusion [I]						
242:Code Injection [CIA]						
248:Command Injection [CIA]	logging	AU-12	AU-12-1	AU-12-3	AU-9	AU-9-3
586:Object Injection [IA]						
594:Traffic Injection [IA]						
74:Manipulating User State [CI]	inputValidation	SI-10	SI-10-5			
	staticCodeAnalysis	SA-11-1				
176:Configuration/Environment Manipulation [I]	memoryProtection	SI-16				
184:Software Integrity Attack [I]	remoteAttestation	IA-3-4				
549:Local Execution of Code [CIA]	secureBoot	SI-7-1	SI-7-5	SI-7-6	SI-7-9	SI-7-15
438:Modification During Manufacture [I]	supplyChainSecurity	SA-12				
439:Manipulation During Distribution [I]	tamperProtection	SA-18-1				

References

1. Siu, K.; Abha, M.; Meng, L.; Michael, D.; Heber, H.-Z.; John, I.; Baoluo, M.; Cesare, T.; Omar, C.; Daniel, L.; et al. Architectural and Behavioral Analysis for Cyber Security. In Proceedings of the 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), San Diego, CA, USA, 8–12 September 2019; pp. 1–10.
2. Papp, D.; Ma, Z.; Buttyan, L. Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy. In Proceedings of the 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, Turkey, 21–23 July 2015.
3. MITRE Corporation. Common Vulnerabilities and Exposures. Available online: https://cve.mitre.org/ (accessed on 12 November 2020).
4. Kordy, B.; Kordy, P.; Mauw, S.; Schweitzer, P. ADTool: Security analysis with attack–defense trees. In Proceedings of the International Conference on Quantitative Evaluation of Systems, Buenos Aires, Argentina, 27–30 August 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 173–176.
5. Pinchinat, S.; Acher, M.; Vojtisek, D. ATSyRa: An integrated environment for synthesizing attack trees. In Proceedings of the International Workshop on Graphical Models for Security, Verona, Italy, 13 July 2015; Springer: Cham, Switzerland, 2015; pp. 97–101.
6. Kwiatkowska, M.; Gethin, N.; David, P. PRISM 4.0: Verification of probabilistic real-time systems. In Proceedings of the International Conference on Computer Aided Verification, Snowbird, UT, USA, 14–20 July 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 585–591.
7. Basagiannis, S.; Katsaros, P.; Pombortsis, A. Probabilistic model checking for the quantification of DoS security threats. *Comput. Secur.* **2009**, *28*, 450–465. [CrossRef]
8. Alexiou, N.; Basagiannis, S.; Katsaros, P.; Dashpande, T.; Smolka, S.A. Formal analysis of the kaminsky DNS cache-poisoning attack using probabilistic model checking. In Proceedings of the 2010 IEEE 12th International Symposium on High Assurance Systems Engineering, San Jose, CA, USA, 3–4 November 2010; pp. 94–103.
9. Celik, Z.B.; McDaniel, P.; Tan, G. Soteria: Automated iot safety and security analysis. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, USA, 18–23 June 2018; pp. 147–158.
10. Manolios, P.; Siu, K.; Noorman, M.; Liao, H. A model-based framework for analyzing the safety of system architectures. In Proceedings of the 2019 Annual Reliability and Maintainability Symposium (RAMS), Orlando, FL, USA, 28–31 January 2019; pp. 1–8.
11. Nieuwenhuis, R.; Albert, O. On SAT modulo theories and optimization problems. In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, Seattle, WA, USA, 12–15 August 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 156–169.
12. Papadopoulos, Y. *HiP-HOPS Automated Fault Tree, FMEA and Optimisation Tool—User Manual*; University of Hull: Hull, UK, 2013.
13. Batteux, M.; Prosvirnova, T.; Rauzy, A. AltaRica 3.0 Language Specification; Altarica Association Report. 2015. Available online: http://www.altarica-association.org/Documentation/pdf/AltaRica%203.0%20Language%20Specification%20-%20v1.2.pdf (accessed on 1 February 2020).
14. Prosvirnova, T.; Batteux, M.; Brameret, P.; Cherfi, A.; Friedlhuber, T.; Roussel, J.; Rauzy, A. The AltaRica 3.0 Project for Model-Based Safety Assessment. In *IFAC Proceedings Volumes*; Elsevier: Amsterdam, The Netherlands, 2013; Volume 46, pp. 127–132.
15. Feller, P.; Hudak, J.; Delange, J.; Gluch, D. *Architecture Fault Modeling and Analysis with the Error Model Annex*, 2nd ed.; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 2016.
16. Delange, J.; Feiler, P. Architecture fault modeling with the AADL error-model annex. In Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications, Verona, Italy, 27–29 August 2014; pp. 361–368.
17. Warg, F.; Skoglund, M. Argument Patterns for Multi-Concern Assurance of Connected Automated Driving Systems. In Proceedings of the 4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019), Stuttgart, Germany, 9–12 July 2019.
18. Taguchi, K.; Souma, D.; Nishihara, H. Safe & sec case patterns. In Proceedings of the International Conference on Computer Safety, Reliability, and Security, Florence, Italy, 10–12 September 2014; Springer: Cham, Switzerland, 2014; pp. 27–37.
19. Martin, H.; Bramberger, R.; Schmittner, C.; Ma, Z.; Gruber, T.; Ruiz, A.; Macher, G. Safety and security co-engineering and argumentation framework. In Proceedings of the International Conference on Computer Safety, Reliability, and Security, Trento, Italy, 12–15 September 2017; Springer: Cham, Switzerland, 2017; pp. 286–297.
20. Sljivo, I.; Gallina, B.; Carlson, J.; Hansson, H.; Puri, S. Tool-supported safety-relevant component reuse: From specification to argumentation. In Proceedings of the Ada-Europe International Conference on Reliable Software Technologies, Lisbon, Portugal, 18–22 June 2018; Springer: Cham, Switzerland, 2018; pp. 19–33.
21. Denney, E.; Pai, G. Tool support for assurance case development. *Autom. Softw. Eng.* **2018**, *25*, 435–499. [CrossRef]
22. Gacek, A.; Backes, J.; Cofer, D.; Slind, K.; Whalen, M. Resolute: An assurance case language for architecture models. *ACM SIGAda Ada Lett.* **2014**, *34*, 19–28. [CrossRef]
23. Cofer, D.; Gacek, A.; Backes, J.; Whalen, M.W.; Pike, L.; Foltzer, A.; Podhradsky, M.; Klein, G.; Kuz, I.; Andronick, J.; et al. A formal approach to constructing secure air vehicle software. *Computer* **2018**, *51*, 14–23. [CrossRef]
24. Hart, E.L. Introduction to Model-Based System Engineering (MBSE) and SysML. 2015. Available online: https://www.incose.org/docs/default-source/delaware-valley/mbse-overview-incose-30-july-2015.pdf (accessed on 1 February 2020).

25. Feiler, P.H.; Gluch, D.P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*; Addison-Wesley: New York, NY, USA, 2012.
26. Cofer, D.; Gacek, A.; Miller, S.; Whalen, M.W.; LaValley, B.; Sha, L. Compositional verification of architectural models. In Proceedings of the NASA Formal Methods Symposium, Norfolk, VA, USA, 3–5 April 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 126–140.
27. Caspi, P.; Pilaud, D.; Halbwachs, N.; Plaice, J. Lustre: A Declarative Language for Programming Synchronous Systems. In Proceedings of the Fourteenth Annual {ACM} Symposium on Principles of Programming Languages, Munich, Germany, 21–23 January 1987.
28. VERDICT Github Repository. Available online: https://github.com/ge-high-assurance/VERDICT (accessed on 1 February 2021).
29. Siu, K.; Herencia-Zapana, H.; Manolios, P.; Noorman, M.; Haadsma, R. Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance; NASA Contractor Report. 2019. Available online: https://ntrs.nasa.gov/citations/20190027401 (accessed on 27 February 2021).
30. Banach, R.; Bozzano, M. Retrenchment, and the generation of fault trees for static, dynamic and cyclic systems. In Proceedings of the International Conference on Computer Safety, Reliability, and Security SAFECOMP, Gdansk, Poland, 27–29 September 2006; pp. 127–141.
31. Safety Analysis with Error Model V2. 6 September 2018. Available online: https://github.com/osate/osate2/blob/master/emv2/org.osate.aadl2.errormodel.help/markdown/safetyanalysis.md (accessed on 23 September 2019).
32. Delange, J.; Hugues, J. Safety Analysis with AADL. 29 September 2015. Available online: http://www.openaadl.org/downloads/tutorial_models15/part5-safety.pdf (accessed on 12 November 2020).
33. Hugues, J.; Delange, J. Model-Based Design and Automated Validation of ARINC653 Architectures Using the AADL. In *Cyber-Physical System Design from an Architecture Analysis Viewpoint*; Springer: Singapore, 2017; pp. 33–52.
34. MITRE. Common Attack Pattern Enumeration and Classification. Available online: https://capec.mitre.org/ (accessed on 16 November 2020).
35. National Institute of Standards and Technology (NIST). *NIST Special Publication (SP) 800-53 Revision 5 (Draft), Security and Privacy Controls for Systems and Organizations*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2017.
36. Moitra, A.; Prince, D.; Siu, K.; Durling, M.; Herencia-Zapana, H. Threat Identification and Defense Control Selection for Embedded Systems. *SAE Int. J. Transp. Cyber. Priv.* **2020**, *3*, 11-03-02-0005. [CrossRef]
37. NIST 800-53 Controls. Available online: https://nvd.nist.gov/800-53/Rev4 (accessed on 1 February 2021).
38. Crapo, A.; Moitra, A. Toward a unified English-like representation of semantic models, data, and graph patterns for subject matter experts. *Int. J. Semant. Comput.* **2013**, *7*, 215–236. [CrossRef]
39. OWL: Web Ontology Language. Available online: https://www.w3.org/OWL/ (accessed on 12 February 2021).
40. Xtext: Language Engineering for Everyone! Available online: http://www.eclipse.org/Xtext/ (accessed on 16 November 2020).
41. Schneier, B. Attack Trees. Available online: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (accessed on 27 February 2021).
42. Jurgenson, A.; Willemson, J. Processing multi-parameter attack trees with estimated parameter values. In *Information and Computer Security*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 308–319.
43. Kordy, B.; Radomirovic, S.; Mauw, S.; Schweitzer, P. Foundations of attack-defense trees. In *Workshop on Formal Aspects of Security and Trust*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 80–95.
44. Kordy, B.; Mauw, S.; Radomirović, S.; Schweitzer, P. Attack–defense trees. *J. Log. Comput.* **2014**, *24*, 55–87. [CrossRef]
45. Siu, K.; Herencia-Zapana, H.; Prince, D.; Moitra, A. A model-based framework for analyzing the security of system architectures. In Proceedings of the Reliability and Maintainability Symposisum, Palm Springs, CA, USA, 31 July 2020.
46. Bayes for Beginners: Probability and Likelihood. Available online: https://www.psychologicalscience.org/observer/bayes-for-beginners-probability-and-likelihood (accessed on 12 November 2020).
47. Javaid, A.; Sun, W.; Devabhaktuni, V.; Alam, M. Cyber Security Threat Analysis and Modeling of an Unmanned Aerial Vehicle System. In Proceedings of the IEEE Conference on Technology for Homeland Security (HST), Waltham, MA, USA, 13–15 November 2012.
48. De Moura, L.; Nikolaj, B. Z3: An efficient SMT solver. In Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
49. Bloomfield, R.; Netkachova, K. Building Blocks for Assurance Cases. In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 15 December 2014.
50. Baoluo, M.; Meng, B.; Moitra, A.; Crapo, A.W.; Paul, S.; Siu, K.; Durling, M.; Prince, D.; Herencia-Zapana, H. Towards Developing Formalized Assurance Cases. In Proceedings of the 2020 IEEE/AIAA 39th Digital Avionics Systems Conference (DASC), San Antonio, TX, USA, 11–15 October 2020.
51. Kelly, T.; Weaver, R. The Goal Structuring Notation—A Safety Argument Notation. In Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases, Florence, Italy, 1 July 2004.
52. Hawkins, R.; Habli, I.; Kolovos, D.; Paige, R.; Kelly, T. Weaving an Assurance Case from Design: A Model-Based Approach. In Proceedings of the IEEE 16th International Symposium on High Assurance Systems Engineering, Daytona Beach Shores, FL, USA, 8–10 January 2015.

53. Champion, A.; Alain, M.; Christoph, S.; Cesare, T. The Kind 2 model checker. In Proceedings of the International Conference on Computer Aided Verification, Toronto, ON, Canada, 17–23 July 2016; Springer: Cham, Switzerland, 2016; pp. 510–517.
54. Burch, J.R.; Clarke, E.M.; McMillan, K.L.; Dill, D.L.; Hwang, L.J. Symbolic model checking: 1020 states and beyond. *Inf. Comput.* **1992**, *98*, 142–170. [CrossRef]