A Methodological Approach to Verify Architecture Resiliency

Joanna C. S. Santos[1], Selma Suloglu[2], Néstor Cataño[2], and Mehdi Mirakhorli[2]

[1] University of Notre Dame, Notre Dame IN 46556, USA
joannacss@nd.edu
[2] Rochester Institute of Technology, Rochester NY 14623, USA
{sxsvse,nxccics,mxmvse}@rit.edu

Abstract. Architecture-first approach to address software resiliency is becoming the mainstream development method for mission-critical and software-intensive systems. In such approach, resiliency is built into the system from the ground up, starting with a robust architecture design. As a result, a flaw in the design of a resilient architecture affects the system's ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on cyber-resources. In this paper, we present an architecture-centric reasoning and verification methodology for detecting design weaknesses in resilient systems. Our goal is to assist software architects in building sound architectural models of their systems. We showcase our approach with the aid of an Autonomous Robot modeled in AADL, in which we use our methodology to uncover three architectural weaknesses in the adoption of three architectural tactics.

Keywords: Cyber resiliency · Architecture Analysis and Design Language · AADL · Architecture Tactics

1 Introduction

Cyber-resiliency refers to a system's ability to anticipate potential compromises, to continue operating even under attacks (*to withstand*), to restore its operation in the face of attacks (*to recover*), and to adapt its behavior to minimize any compromises (*to evolve*) [5]. Achieving cyber-resiliency goals, therefore, involves designing a software system that addresses multiple quality attributes, such as performance, security, safety, or evolvability. Under these circumstances, the architecture-first development method [7] is becoming the mainstream approach for addressing cyber resiliency concerns in mission-critical and software-intensive systems [10,14]. Since the system's architecture design plays a crucial role in the software development process, weaknesses in the software system's architecture can have a greater impact on the system's ability to anticipate, withstand, recover from, and adapt to adverse conditions, stresses, attacks, or compromises on cyber resources.

Despite the importance of the architecture-first approach to enhance and ensure the resiliency of mission-critical systems, current research in the field focuses on the verification of requirements, functional or non-functional (e.g., safety, or security) of the system [31,16,1], and neglects an in-depth analysis of architectural weaknesses in the system's design. Some existing approaches for creating architectural models comply with domain-specific requirements (e.g., avionics) [27,31,20,6], or with the creation of reusable modeling components [17,16]. However, cyber-resiliency involves multiple quality attributes (availability, safety, security, etc.) and can be applied to multiple systems domains. Finally, the current state of practice requires systems engineers to have an in-depth understanding of potential mistakes associated with the design of resilient systems and use qualitative techniques to evaluate the design [22]. For instance, the Architecture Trade-off Analysis Method (ATAM) [22] has been widely used in mission and safety critical applications as a qualitative approach to risk and trade-off analysis of an architecture with respect to a set of clearly articulated quality scenarios. However, such approaches are not able to detect specific design weaknesses in complex applications with several components and interdependencies.

In this paper, we describe ***an architecture-centric reasoning and verification methodology*** for detecting design weaknesses in resilient systems. It is a model-driven methodology for aiding software architects to systematically analyze and formally verify the resiliency aspects of their architectural designs. It shifts the architecture evaluation from a primarily qualitative and subjective approach to an approach that is empowered by formal verification.

Our methodology encompasses four phases. ① The architecture is modelled using the Architecture Analysis and Design Language (AADL) [12]. ② The architecture model is enriched with *annotations* that map AADL components to elements in resiliency *tactics* and *patterns*. These annotations add semantics of resiliency tactics and patterns to AADL models, allowing reasoning about flaws associated to a resilient design. ③ A risk assessment is performed to identify *weaknesses* that may violate the properties of resiliency tactics in the system. In this phase, these weaknesses are specified in terms of *conceptual models* and a set of *formal rules*. These rules are written using the *Resolute* [13] language, which is application-independent and can be reused to analyze another system modeled in AADL. ④ Finally, these rules are checked against the annotated model to verify whether the current architecture design is flawed (i.e., it contains architectural weaknesses). We showcase our methodology with the aid of an Autonomous Robot, for which we model its system architecture using the Architectural Analysis & Design Language (AADL) [12]. We use Resolute to specify three common architectural weaknesses to expose flaws in the Autonomous Robot's design.

The contributions of this paper are three-fold. (*i.*) a novel methodology for the verification of architectural resiliency properties of AADL models. (*ii.*) An approach to specify common architectural weaknesses as conceptual models which are converted into reusable rules written in an assurance case language (Resolute), which can be used to detect design weaknesses in various architectures. We use the *Resolute* [13] language for pragmatic reasons. This language was initially developed to model assurance cases, however, we adopt it because of its power to conduct reachability analysis of AADL models which can be leveraged

to detect various design weaknesses. (*iii.*) A case study demonstrating the feasibility and practicality of using our methodology to detect common architectural weaknesses in complex AADL models.
Paper organization. Section 2 introduces concepts for our paper to be understood by a broader audience. Section 3 describes our model-driven methodology. Section 4 illustrates the methodology in the context of an autonomous robot. Section 5 discusses related work, whereas Section 6 concludes this paper.

2 Background

This section explains key concepts that are used throughout the paper.

2.1 Architectural Tactics

Software architects typically use a rich set of proven architectural *tactics* to design cyber-resilient systems [2]. They provide reusable solutions for addressing resiliency concerns, even when the system is under attack. They are grouped under five main categories: *detect*, *resist* (withstand), *react to*, *recover from*, and *prevent* cyber events [2,5,18]. Tactics play an important role in shaping the high-level design of software since they describe reusable techniques and concrete solutions for satisfying a wide range of quality concerns [2,15], including resiliency.

The Software Engineering Institute at Carnegie Mellon University released a comprehensive catalog of tactics for different quality attributes such as availability, security, and reliability [2]. This catalog is collected from existing literature. Besides, there is an extensive body of work about the importance of architectural tactics and their role in software quality [2,26].

There are many kinds of resiliency tactics. For example, a system with high reliability requirements might implement the *heartbeat* tactic [2,26] to monitor the availability of a critical component, or the voting tactic [26] to increase fault tolerance through integrating and processing information from a set of redundant components. Architectural tactics are pervasive in resilient and fault-tolerant systems [25,2].

2.2 Common Architectural Weaknesses

Although tactics provide well-formed strategies for a system to satisfy a specific quality concern, e.g., resist cyberattacks, if they are not carefully adopted in a system, they can result in *architectural weaknesses* [19,28]. There is a fundamental difference between *architectural weaknesses* and *bugs*. While the latter are more code-level, such as buffer overflows caused by miscalculations, the former are at a higher level and much more subtle and sophisticated [19].

For instance, a system may adopt the *Authenticate Actors* tactic, but the authentication enforcement is performed in the client-side instead of the server-side [28]. In this example, a client/server product performs authentication within the client code, but not in the server code, allowing the authentication feature to be bypassed via a modified client which omits the authentication check. This design decision creates a weakness in the security architecture, which can be successfully exploited by an intruder with reverse-engineering skills. There are numerous examples of architectural weaknesses (also referred in the literature as "design flaws") [28,21,33,19] that needs to be mitigated at the design time.

2.3 Architecture Modeling and AADL

The Architecture Analysis and Design Language (AADL) [12] is a modeling language which allows an architecture-centric, and model-based development approach throughout the system lifecycle. AADL models cover both *static* and *dynamic* system structure in terms of *components* and their *interactions*. Components are categorized as either *application system* components or *execution platform* components (as shown in Figure 1). Components include a set of *features*, *properties*, and *flows*.

Application system components interact with each other via *features* (one or more ports or data accesses). Features that are either data and/or event ports can be an input (*in*), output (*out*) or input/output (*in/out*) port. The interaction between components in an execution platform is provided by a *Bus* component type, which is the counterpart of a feature in the application system components.

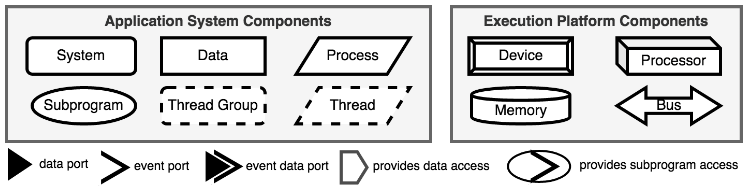


Fig. 1: AADL graphical notation

AADL also allows components to be annotated with *properties*, which define the characteristics of a component. These properties can be used to perform various analysis, such as model checking. AADL provides six predefined *property sets* but AADL models can also be extended with user-defined property sets.

2.4 Architecture Reasoning on AADL using Resolute

Given that an AADL model provide a formalized presentation of the architecture, we can leverage its semantics to perform reasoning. For this purpose, *Resolute* [13] is a language and a tool to assure the system's architecture specified in AADL meet its expectations, which are represented as a set of rules expressed in a declarative fashion. These rules are written in terms of *assurance cases*, each of which presents *claims* about the system and supporting *evidence* for them. These claims describe a quality attribute about the system (e.g., safety cases, security cases, etc.). The evidence for the claim is automatically extracted from the model based on the written rules.

For instance, one can claim that the system logs all security-relevant operations (i.e., it adopts the *Maintain Audit Trails* tactic [2]). As shown in Listing 1.1, this can be done by first defining a top-level claim (line 5) that describes using a first-order predicate how this claim can be proved. This check is implemented by querying the AADL model to verify that the system has (i) at least one Audit Manager, which is a component that tracks each transaction alongside with identifying information (line 7); and (ii) at least one Action Target, which is a component that perform security-critical operations (line 8). This top-level claim invokes a subclaim (`all_action_targets_report_operations(s)`[3] in line 9) to check whether all Action Targets do send their critical operation information to the Audit Manager for it to be logged.

```
 1  annex resolute {**
 2    has_role(s: aadl, role: string) :
 3      bool = (has_property (s, Props::Role)) and member(role, property(s, Props
            ::Role))
 4
 5    maintain_audit_trails(s: system) <=
 6      ** "The system " s " logs security-relevant operations" **
 7      exists(comp: component) . has_role(comp, "AuditManager") and
 8      exists(comp: component) . has_role(comp, "ActionTarget") andthen
 9      all_action_targets_report_operations(s)
10
11    -- more subclaims (...)
12  **};
```

Listing 1.1: Claims in Resolute

3 Architecture-centric Verification Methodology

As shown in Figure 2, the architecture-centric reasoning and verification methodology presented in this paper has four phases: ① An ***architecture modeling*** phase in which the software architect models the architectural design using AADL. ② A ***common architectural weakness modeling and specification*** phase in which the software architect selects a potential architectural flaw for inspection, draws a conceptual graph of the flaw, and converts it into a set of *reusable rules*, which are written in Resolute [13]. They describe the way the system should withstand to overcome the flaw. ③ A ***model annotation*** phase in which elements in the architectural model are tagged with *roles* and *properties*. A role describes a component's functionality, whereas a property defines the characteristics of a component. ④ An ***architecture verification*** phase, in which the architectural elements (components) tagged during the third phase are checked against the Resolute rules encoded during the second one. The Resolute tool checks for (the absence of) weaknesses automatically.

[3] Due to space constraints, we do not show the definitions for all sub-claims

These four phases are performed repeatedly until the software architecture achieves the intended level of resiliency. In what follows, we detail each phase.

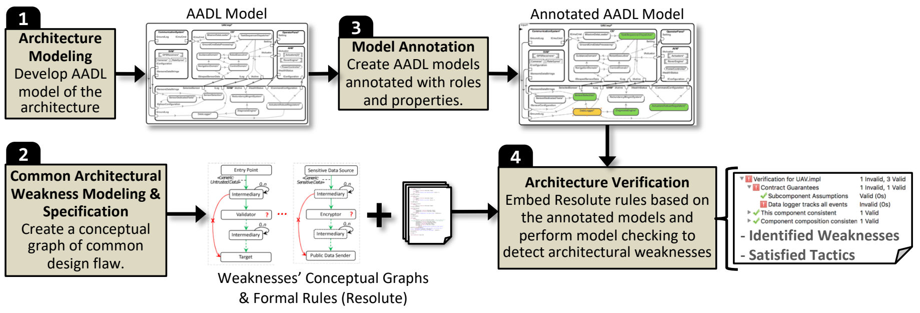

1
Architecture Modeling
Develop AADL model of the architecture
AADL Model
3
Model Annotation
Create AADL models annotated with roles and properties.
Annotated AADL Model
2
Common Architectural Weakness Modeling & Specification
Create a conceptual graph of common design flaw.
Weaknesses' Conceptual Graphs & Formal Rules (Resolute)
4
Architecture Verification
Embed Resolute rules based on the annotated models and perform model checking to detect architectural weaknesses
Verification for UAV.impl 1 Invalid, 3 Valid
Contract Guarantees 1 Invalid, 1 Valid
Subcomponent Assumptions Valid (0s)
Data logger tracks all events Invalid (0s)
This component consistent 1 Valid
Component composition consisten 1 Valid
- Identified Weaknesses
- Satisfied Tactics

Fig. 2: The architecture-centric reasoning and verification methodology

3.1 Architecture Modeling

In this phase, the software architect uses AADL to describe system components, their interactions, as well as functional and non-functional properties of the software system. The software architect can use *property sets* and the *annexes* in AADL to specify behavioural details of the design, define the system's components (e.g., threads, subprograms, etc) or the execution platform components (e.g., device, memory, etc). For medium to large scale systems, these models are often non-trivial and may have several elements and inter-dependencies.

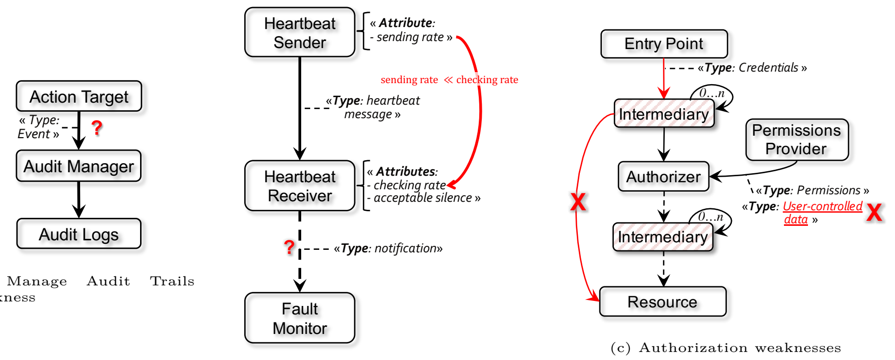
3.2 Common Architectural Weakness Modeling & Specification

In the second phase, the architect performs an assessment of *potential weaknesses*, which are captured in **Weaknesses Models** and **Specifications**.

Modeling Common Architectural Weaknesses. A ***weakness model*** can be used as a guideline for manual or automated inspection of a system's architecture. A weakness model is essentially a *reusable conceptual graph* centered around a specific *architectural tactic* and includes the following elements:

– A set of **architectural roles**, which represent the functionality assumed by an architectural element. These roles are modeled as ***nodes*** in the conceptual graph.
– A set of **properties** that are used to model how the system incorrectly behaves in the occurrence of a cyber-event. They correspond to the data flow and the control flow between the conceptual roles in a Weakness Model. Unlike a data flow, a control flow is a flow that only occur if a condition is true. These data and control flow are ***directed edges*** in the conceptual

graph. Edges have an attribute that specifies the *data type* flowing between the nodes. This attribute could be either *generic*, which means that its data type is not specified or matches all data types, or *typed*, which indicates its actual concrete data type.



(a) Manage Audit Trails weakness

(b) Heartbeat weaknesses

(c) Authorization weaknesses

Fig. 3: Conceptual graphs

Capturing architectural weaknesses as conceptual graphs makes the Weakness Model ***language-agnostic*** and ***reusable***. They can be used to guide the process of detecting architectural weaknesses in systems designs using different modeling languages such as AADL, SysML (Systems Modeling Language), among others. Although the modeling of weaknesses via conceptual graphs can be skipped, it helps the later development of weaknesses specifications, as it demonstrates how the flaw can happen.

Weakness Model Examples. Figure 3 contains examples of conceptual graphs for three weaknesses. Conditional flows, that is, data that only flows if a condition holds, are modeled as dashed arrows whereas data flows are modeled as full arrows in these conceptual graphs. Edges have an attribute that specifies the *data type* flowing between the nodes. This attribute could be either *generic*, which means that its data type is not specified or matches all data types, or *typed*, which indicates the actual data type. For instance, in Figure 3c the Entry Point's returned data is of *credential* type and the data flowing from the Permissions Provider to the Authorizer node is of *user-controlled data* type (which is the root cause of the authorization bypass).

In the first example, shown in Figure 3a, it depicts a weakness related to the "Manage Audit Trails" tactic [2], which can be adopted to log activities in the system for achieving non-repudiation goals and help with system recovery. In this tactic, Action Target components, that perform critical operations, report to the Audit Manager any critical operation being performed alongside with who made that request (i.e., the actor) such that the manager can record that operation in

Audit Logs. The conceptual graph in Figure 3a shows a weakness when the system does not record important activities within their logs because an Action Target is not sending these logs to the Audit Manager to be recorded. This is represented by a question mark that denotes a missing expected interaction between the components.

Figure 3b shows a conceptual model for two weaknesses associated with the "Heartbeat" architectural tactic [2,23], which is used for addressing reliability and availability goals. This tactic encompasses three roles: a safety-critical Heartbeat Sender component that periodically sends *heartbeat messages* to a Heartbeat Receiver to notify that it is still alive. The Heartbeat Receiver is able to detect failures in the safety-critical component when it notices that heartbeat messages are not being received. The Fault Monitor takes action upon detected failures. A potential weakness to this tactic occurs when the *heartbeat checking rate* outpaces the *sending rate* because the Heartbeat Receiver will mistakenly assume that the sender failed. A second weakness is caused by not notifying the Fault Monitor when a failure is detected. Figure 3b shows a conceptual graph modeling these two weaknesses (represented by an edge indicating the rates mismatches between the Sender and Receiver as well as a question mark indicating the lack of notification).

Figure 3c shows two weaknesses when adopting the "Authorize Actors" tactic [2,28]. This tactic enforces actors to hold certain privileges to access any resource that might require them. This Weakness Model has six distinct roles: the system's Entry Point (which is fed with user's inputs), multiple Intermediary nodes (any component transferring data), the Authenticator (used to check the user's identity), the Authorizer (which verifies the actor's permissions against the permissions given by the Permissions Provider), and the Resource node which is accessible to the user. These two weaknesses result in authorization bypasses. A path from an Entrypoint to a Resource without going through the Authorizer component (depicted with a red edge) is the root cause of the first bypass. Relying on user-controlled data to perform the authorization checks is the root cause of the second bypass. This second bypass is highlighted with a font colored in red for the data type provided to the Permissions Provider.

Specifying Common Architectural Weaknesses Formally. Conceptual graphs are good at depicting the information flow between architectural components. However, we need to use a language that allows us to conduct a reachability analysis across the system's components. This is realized as a set of rules in the Resolute language that describe how the system can withstand flaws (weaknesses). This reachability analysis is conducted with the Resolute tool. Creating the Resolute rules for specifying architectural weaknesses involves three activities [29]: **(1)** Writing a ***custom AADL property set*** [12] that declares roles, attributes, and data types; **(2)** Developing ***computation functions*** for checking/querying properties in the AADL model; and **(3)** Writing ***claims*** specifying the system behavior under certain conditions [13].

Weakness Specification Example. Figure 4a presents an example of *property set* definition in Resolute. It introduces role types such as `Resource`, `Entrypoint`, `Authorizer`, etc. The specification states that components such as "thread", "subprogram", "process", among others, hold that type. These roles and role types can be used by software architects to define custom *computation functions* with the aid of annexes, as shown in Figure 4b. The `has_role(s)` computation function returns true if the *s* AADL component contains the given role. The `get_all(role)` computation function returns all the AADL components with a specific role.

```
property set Props is
  RoleType: type enumeration
        (Resource, Entrypoint, Authorizer, …);
  Role: list of Props::RoleType
        applies to (thread, subprogram, process, system, …);
  DataType: enumeration (Credential, HeartbeatMessage, …)
            applies to (data, data port, event data port, …);
  (…)
end Props;
```

(a) Custom property set

```
annex resolute {**
  has_role(s: aadl, role: string):
     bool = has_property(s, Props::role) and
            member(role, property(s, Props::Role))
  get_all(role: string):
     {component} = {y for (x: component)
                       (y: x) | has_role(x,role)}
  (…)
**};
```

(b) Examples of Computation functions

Fig. 4: Specification Rules for an Architectural Weakness

Once a set of computations is declared, the architect proceeds to develop structured Resolute *claims* that describe the architecture formally. For instance, for the conceptual graph in Figure 3b, one can write structured claims as shown in Figure 5, namely, (i) the system has the three required roles for the tactic (Sender, Receiver and Monitor), (ii) the safety-critical components send *heartbeat messages* to the receivers, (iii) the receivers periodically check whether the safety-critical component(s) are still functioning, (iv) the receivers notify the fault monitor upon a failure detection. Due to space constraints, we only show some high-level claims for enforcing the aforementioned properties.

```
check_heartbeat(s: system) <=
  ** "The system " s " adopts Heartbeat tactic"
  " to detect faults in critical components" **
  system_has_role(s,"HeartbeatReceiver") and
  system_has_role(s,"HeartbeatSender") and
  system_has_role(s,"FaultMonitor") and
  critical_components_sends_heartbeat(s) andthen
  receivers_checks_periodically(s) andthen
  receivers_notifies_monitors(s)
```

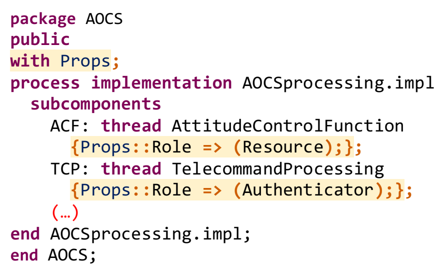
```
system_has_role(s: system, role: string) <=
  ** "The system " s " has a " role " component" **
  exists(comp: component) . has_role(comp, role)

critical_components_sends_heartbeat(s: system) <=
  ** " All the senders periodically send a heartbeat" **
  forall (sender : get_all("HeartbeatSender")) .
    exists(receiver : get_all("HeartbeatSender")).
    sends_heartbeat(sender, receiver) and
    property(sender, Props::SendingRate) =
    property(receiver, Props::CheckingRate)
```

Fig. 5: High-level claims for the Heartbeat tactic

3.3 Architectural Model Annotation

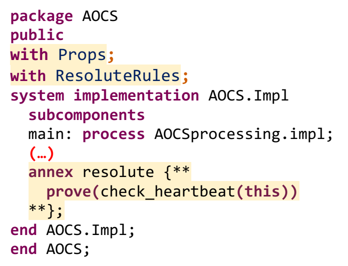
In this third phase, the architectural model is annotated with metadata about resiliency tactics. The metadata created in this phase is crucial to indicate which components implement resiliency tactics, and express the expected characteristics and behavior of the system. To perform these annotations, the architects first add an import statement into the system's AADL file to import the previously developed files that contain the custom property set, claims and computations.

Then, the elements in the AADL model are annotated with an *architectural role* of components, their *attributes* and *data types*. Figure 6a shows an example of an architectural model of an Attitude and Orbit Control System (AOCS) [8], in which the `TCP` component is annotated with the Authorizer role to indicate that it is in charge of checking the privileges of actors interacting with the system whereas the `ACF` component is the target Resource (i.e., the component that needs to be protected against unauthorized access) [29].



```
package AOCS
public
with Props;
process implementation AOCSprocessing.impl
  subcomponents
    ACF: thread AttitudeControlFunction
      {Props::Role => (Resource);};
    TCP: thread TelecommandProcessing
      {Props::Role => (Authenticator);};
    (…)
end AOCSprocessing.impl;
end AOCS;
```

(a) An annotated AADL model



```
package AOCS
public
with Props;
with ResoluteRules;
system implementation AOCS.Impl
  subcomponents
  main: process AOCSprocessing.impl;
  (…)
  annex resolute {**
    prove(check_heartbeat(this))
  **};
end AOCS.Impl;
end AOCS;
```

(b) Inserting Resolute rules into an AADL model

Fig. 6: AADL Model Annotation and Rules Embedding Examples

3.4 Architecture Verification

During the final phase, the Resolute rules are added to the top-level implementation of the system. This is done by invoking the resolute claims using the `prove` keyword. For instance, Figure 6b shows the inclusion of the Resolute rules to the top-level system implementation by using `prove(check_heartbeat(this)`. When Resolute executes, it will verify the claims over the current system implementation (i.e., `AOCS.Impl`).

These claims can be checked using the Resolute plug-in installed on the Open Source AADL Tool Environment (OSATE) [11]. This plugin will then conduct a soundness proof over the entire system. Therefore, the proof is replicated (valid) for very instance of the component's implementation.

4 Case Example

We illustrate our approach with the aid of an autonomous robot based on the resources provided by the NASA Lunar Robot [24]. The robot's primary mission is to autonomously traverse the lunar surface, collect sample data related to comets, dust, and celestial objects, record temperatures, perform scientific experiments, and send results back to the earth-based Mission Control Center.

– **Phase 1: Modeling the Robot Architecture** The Robot, whose architecture is shown in Figure 7, has the capability of operating in two modes: *manual* (remotely controlled by a ground control station) and *autonomous* (by following a pre-established flight plan). Its architecture is structured

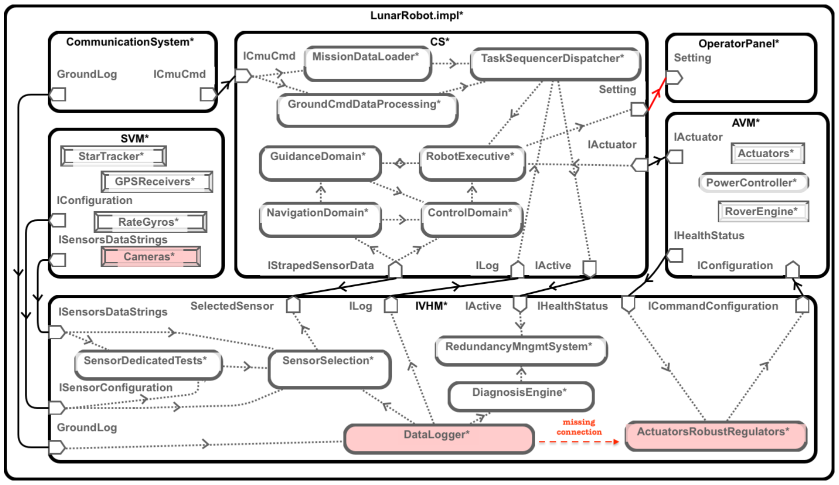


Fig. 7: High-Level Architecture Model of the Lunar Robot in AADL

around a *Control System* (CS), an *Integrated Vehicle Health Management* (IVHM) system, the *Sensors Virtual Machine* (SVM), an *Actuators Virtual Machine* (AVM), a *Communication System*, and an *Operator Panel*. The *Communication System* implements the communication protocol to receive commands sent from the ground station and forwarding those to the *Control System* (CS). Data from the sensors is first passed through the IVHM component for correctness checking and is then forwarded to the CS. The CS uses the data to make decisions and to process high-level commands sent by the ground station. It then sends lower level commands to the actuators. The *Integrated Vehicle Health Management* (IVHM) is responsible for monitoring the health of the Lunar Robot, and when necessary, performing dynamic reconfigurations to maintain functionality. The Lunar Robot receives inputs from cameras, GPS receivers, rate gyros, and star trackers, and issues command to mechanical devices such as the power controller, wheels, and scientific instruments.

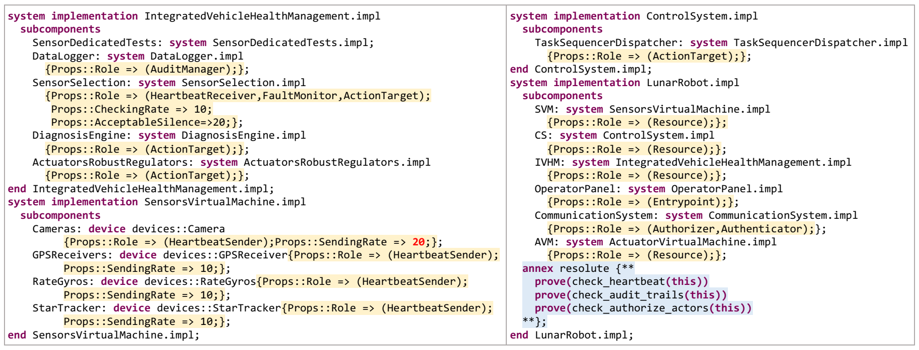
Resiliency Requirements & Adopted Tactics: To achieve resiliency, the lunar robot's design adopts the following tactics:

- **Heartbeat Tactic**: the sensors within the "SVM" component periodically heartbeat messages to the "Sensors Selection" component, which acts as both a Heartbeat Receiver and a Fault Monitor.
- **Authorize Actors**: the "Communication System" checks the identity and privileges of the Mission Control Center (MCC) before exchanging data and accepting commands from it.
- **Maintain Audit Trails**: Whenever the robot is in manual mode, but receives no commands from the MCC for an extended period of time, it switches to autonomous mode and returns to the geographical coordinates of the last *point of known contact* (which is recorded in data logs).

Given this requirement, the system adopts the *Maintain Audit Trails* tactic [2] for maintaining logs that can later be used for system recovery.

For this case example, we limit our discussion to three tactics, however, in medium to large scale systems, a variety of tactics are adopted.

– **Phase 2: Architectural Weaknesses Modeling & Specification** Subsequently, the architect identifies potential weaknesses while adopting the aforementioned tactics. First, the architect creates conceptual graphs for these weaknesses, as previously shown in Figure 3 for the "Maintain Audit Trails", "Heartbeat", and "Authorize Actors" tactics. Given these conceptual graphs, the software architect creates a property set declaring tactics' role types, and data properties, as previously shown in Figure 4a. Using this property set file, resolute rules are written to formally verify that these weaknesses do not occur in the current design (e.g., Figure 5). Notice that these models and specifications are written in such a way that make them ***reusable***. As a result, architects could re-use these models and specifications to verify the same weaknesses in other designs that also adopt these tactics, reducing the efforts in creating these artifacts from scratch.



```
system implementation IntegratedVehicleHealthManagement.impl
  subcomponents
    SensorDedicatedTests: system SensorDedicatedTests.impl;
    DataLogger: system DataLogger.impl
      {Props::Role => (AuditManager);};
    SensorSelection: system SensorSelection.impl
      {Props::Role => (HeartbeatReceiver,FaultMonitor,ActionTarget);
       Props::CheckingRate => 10;
       Props::AcceptableSilence=>20;};
    DiagnosisEngine: system DiagnosisEngine.impl
      {Props::Role => (ActionTarget);};
    ActuatorsRobustRegulators: system ActuatorsRobustRegulators.impl
      {Props::Role => (ActionTarget);};
end IntegratedVehicleHealthManagement.impl;
system implementation SensorsVirtualMachine.impl
  subcomponents
    Cameras: device devices::Camera
         {Props::Role => (HeartbeatSender);Props::SendingRate => 20;};
    GPSReceivers: device devices::GPSReceiver{Props::Role => (HeartbeatSender);
         Props::SendingRate => 10;};
    RateGyros: device devices::RateGyros{Props::Role => (HeartbeatSender);
         Props::SendingRate => 10;};
    StarTracker: device devices::StarTracker{Props::Role => (HeartbeatSender);
         Props::SendingRate => 10;};
end SensorsVirtualMachine.impl;
```

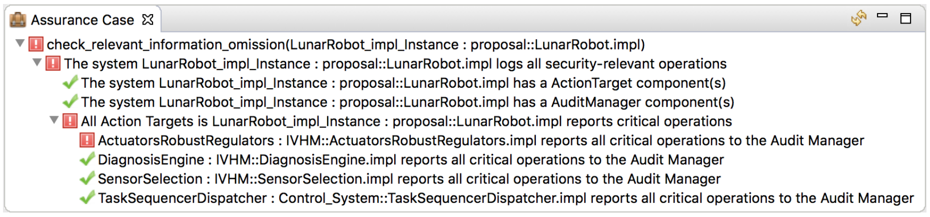
```
system implementation ControlSystem.impl
  subcomponents
    TaskSequencerDispatcher: system TaskSequencerDispatcher.impl
      {Props::Role => (ActionTarget);};
end ControlSystem.impl;
system implementation LunarRobot.impl
  subcomponents
    SVM: system SensorsVirtualMachine.impl
      {Props::Role => (Resource);};
    CS: system ControlSystem.impl
      {Props::Role => (Resource);};
    IVHM: system IntegratedVehicleHealthManagement.impl
      {Props::Role => (Resource);};
    OperatorPanel: system OperatorPanel.impl
      {Props::Role => (Entrypoint);};
    CommunicationSystem: system CommunicationSystem.impl
      {Props::Role => (Authorizer,Authenticator);};
    AVM: system ActuatorVirtualMachine.impl
      {Props::Role => (Resource);};
  annex resolute {**
    prove(check_heartbeat(this))
    prove(check_audit_trails(this))
    prove(check_authorize_actors(this))
  **};
end LunarRobot.impl;
```

Fig. 8: Lunar Robot's AADL model annotated with roles

– **Phase 3: Model Annotation** Once weaknesses' models and specifications are created, architects manually tag the Lunar Robot's architecture with roles and properties, as shown in Figure 8. This figure does not contain all the AADL model, but just the elements that have annotations.

– **Phase 4: Architecture Verification**
The architect inserts the developed resolute rules (highlighted in blue in Figure 8). Through performing a reasoning on top of this augmented model (containing analysis rules and annotations), the technique detects the following weaknesses (whose locations were colored in red in Figure 7):
Weakness #1: an *Omission of Security-relevant Information* because the *Data Logger* is not tracking information from the actuators. It is caused by a connection missing from the *Data Logger* to the *ActuatorsRobustRegulators*.

Figure 9 shows the output of the resolute tool indicating this problem (notice the failed claim that all Action Targets are reporting critical operations to the Audit Manager).
Weakness #2: a *Mismatch between* Send *and* Receive *Periods* which is caused by the *Cameras* sending heartbeat messages every 20 seconds whereas the checking rate of the "Sensor Selection" (Heartbeat receiver) is 10 seconds, as indicated in the *SendingRate* value colored in red in Figure 8.
Weakness #3: an *Authorization bypass* because the "Operator Panel" component, which is also an entrypoint to the system, directly communicates with the "Control System" (CS) without authorization (connection highlighted in red in Figure 7).



```
▼ ! check_relevant_information_omission(LunarRobot_impl_Instance : proposal::LunarRobot.impl)
  ▼ ! The system LunarRobot_impl_Instance : proposal::LunarRobot.impl logs all security-relevant operations
      ✓ The system LunarRobot_impl_Instance : proposal::LunarRobot.impl has a ActionTarget component(s)
      ✓ The system LunarRobot_impl_Instance : proposal::LunarRobot.impl has a AuditManager component(s)
    ▼ ! All Action Targets is LunarRobot_impl_Instance : proposal::LunarRobot.impl reports critical operations
        ! ActuatorsRobustRegulators : IVHM::ActuatorsRobustRegulators.impl reports all critical operations to the Audit Manager
        ✓ DiagnosisEngine : IVHM::DiagnosisEngine.impl reports all critical operations to the Audit Manager
        ✓ SensorSelection : IVHM::SensorSelection.impl reports all critical operations to the Audit Manager
        ✓ TaskSequencerDispatcher : Control_System::TaskSequencerDispatcher.impl reports all critical operations to the Audit Manager
```

Fig. 9: Resolute output

5 Related Work

There are several streams of research on analyzing and verifying architectural models by translating AADL models into model checkers [20,6]. One major difference between these works and our work is the property notation used for specifying behavior; the prior work uses subsets of the AADL behavioral annex to focus on low level implementations issues. In contrast, we focus on design flows. Johnsen et al. [20] presents a technique for verifying the completeness and consistency of AADL specifications.

Furthermore, there have been various works which use ADDL along with a declarative formalism provided by languages such as AGREE, to verify the satisfaction of software requirements in the model [1]. However, these approaches will not address model checking of components and interdependencies among them, as well as specific resiliency properties. In contrast, we rely on Resolute that enables reachability analysis required to examine the dependencies between components and elements in AADL to detect various bypasses of mitigation techniques. For instance, the SPEEDS [1] approach uses a model-based engineering methodology supported by formal analyses. However, it focuses on verification of software requirements. It requires that the architectural description of the system (AADL models) to be annotated with assume-guarantee style contracts on components that implement software requirements.

Ellison et al. [9] proposes an enhancement for the AADL to incorporate security concerns based on Microsoft STRIDE framework. Similar to our work, they discuss a custom property set which is used to annotate and analyze the system architecture. Our work, however, differs in that we provide a *methodology* whereby we provide a continuous resiliency analysis starting with the system's architecture instead of a threat model.

Prior works also explored the automated detection of weaknesses in dataflow diagrams (DFDs) [32,3,30]. For example, Berger et al. [4] presented a tool-supported approach to help architectural risk analysis via threat modeling; the tool is used to automatically identify security threats from data flow diagrams. Tuma et al. [32] described the use of reusable queries to find security flaws in DFDs. Unlike these prior works, our paper describes a *methodology* to systematically find weaknesses in the system that affects its resiliency, that include not only security concerns, but also other quality attributes, such as safety, performance, and availability.

6 Conclusion

We have presented a methodological approach to detect the existence of common architectural resiliency weaknesses of systems. At the core of this methodology is the development of weaknesses models and specifications, which are used to automatically verify the system's resiliency. Although the construction of these models and specifications rely on an architects expertise, they can be built in such a way that can be reused across AADL models for different systems.

Our long-term goal is to provide software architects with mechanisms and tools to develop *certified* software (and hardware) all the way down from architectural design to implementations details. This work is the first part of our envisaged work. As future work, we plan to automate the verification process with the OSATE IDE [11] and Resolute tools [13]. We plan to create a complete catalog of resiliency tactics, write a comprehensive set of assurance cases for each tactic, and automate the process of design and checking of resiliency claims through the realization of an Osate plug-in.

Acknowledgement

This work is partially supported by Defense Advanced Research Projects Agency (DARPA) under award number: 006376-002.

References

1. SPEculative and Exporatory Design in System engineering, `http://www.speeds.eu.com/`
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional, 3rd edn. (2012)

3. Berger, B.J., Sohr, K., Koschke, R.: Extracting and analyzing the implemented security architecture of business applications. In: 17th European Conference on Software Maintenance and Reengineering (CSMR). pp. 285–294. IEEE (2013). https://doi.org/10.1109/CSMR.2013.37
4. Berger, B.J., Sohr, K., Koschke, R.: Automatically extracting threats from extended data flow diagrams. In: International Symposium on Engineering Secure Software and Systems. pp. 56–71. Springer (2016)
5. Bodeau, D., Graubart, R.: Cyber resiliency design principles. MITRE (2017)
6. Bodeveix, J.P., Filali, M., Garnacho, M., Spadotti, R., Yang, Z.: Towards a verified transformation from aadl to the formal component-based language fiacre. Science of Computer Programming **106**, 30 – 53 (2015)
7. Booch, G.: The economics of architecture-first. IEEE Software **24**(5), 18–20 (Sep 2007). https://doi.org/10.1109/MS.2007.146
8. Cechticky, V., Montalto, G., Pasetti, A., Salerno, N.: The AOCS framework. European Space Agency-Publications-ESA SP **516**, 535–540 (2003)
9. Ellison, R., Householder, A., Hudak, J., Kazman, R., Woody, C.: Extending aadl for security design assurance of cyber-physical systems. Tech. Rep. CMU/SEI-2015-TR-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2015), `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=449510`
10. Feiler, P.H., Gluch, D., McGregor, J.D.: An architecture-led safety analysis method. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016) (2016)
11. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley (2012)
12. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis & design language (AADL): An introduction. Tech. rep., SEI (2006). https://doi.org/10.1184/R1/6584909.v1
13. Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M.: Resolute: An assurance case language for architecture models. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 19–28. ACM, New York, NY, USA (2014)
14. Goldman, H.G.: Building secure, resilient architectures for cyber mission assurance. Tech. rep., The MITRE Corporation (2010)
15. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley Series in Software Design Patterns (2007)
16. Heyman, T., Scandariato, R., Joosen, W.: Reusable formal models for secure software architectures. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA). pp. 41–50. IEEE (2012)
17. Hugues, J.: AADLib: a library of reusable AADL models. Tech. rep., SAE Technical Paper (2013)
18. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale. arXiv preprint arXiv:1708.07422 (2017)
19. IEEE Center for Secure Design: Avoiding the top 10 software security design flaws. `https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf` (2015), (Accessed on 10/06/2016)
20. Johnsen, A., Lundqvist, K., Pettersson, P., Jaradat, O.: Automated verification of aadl-specifications using uppaal. In: Proceedings of the 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering. p. 130–138. HASE '12, IEEE Computer Society, USA (2012)

21. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A.: A case study in locating the architectural roots of technical debt. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2, pp. 179–188 (2015)
22. Kazman, R., Klein, M., Clements, P.: Atam: A method for architecture evaluation. Software Engineering Institute (2000)
23. Kim, S., Kim, D.K., Lu, L., Park, S.Y.: A tactic-based approach to embodying non-functional requirements into software architectures. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference. pp. 139–148 (2008)
24. Mirakhorli, M., Cleland-Huang, J.: Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance. pp. 123–132. ICSM '11, IEEE Computer Society, Washington, DC, USA (2011)
25. Mirakhorli, M., Cleland-Huang, J.: Detecting, tracing, and monitoring architectural tactics in code. IEEE Transactions on Software Engineering **42**(3), 205–220 (2015)
26. Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.: A tactic-centric approach for automating traceability of quality concerns. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, IEEE Press (2012)
27. Munoz, M.: Space systems modeling using the architecture analysis & design language (AADL). In: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 97–98. IEEE (2013)
28. Santos, J.C.S., Tarrit, K., Mirakhorli, M.: A catalog of security architecture weaknesses. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 220–223 (April 2017)
29. Santos, J.C.S., Suloglu, S., Ye, J., Mirakhorli, M.: Towards an Automated Approach for Detecting Architectural Weaknesses in Critical Systems, p. 250–253. Association for Computing Machinery, New York, NY, USA (2020), `https://doi.org/10.1145/3387940.3392222`
30. Sion, L., Tuma, K., Scandariato, R., Yskout, K., Joosen, W.: Towards automated security design flaw detection. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). IEEE (2019)
31. Stewart, D., Whalen, M.W., Cofer, D., Heimdahl, M.P.: Architectural modeling and analysis for safety engineering. In: International Symposium on Model-Based Safety and Assessment. pp. 97–111. Springer (2017)
32. Tuma, K., Sion, L., Scandariato, R., Yskout, K.: Automating the early detection of security design flaws. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 332–342 (2020)
33. Vanciu, R., Abi-Antoun, M.: Finding architectural flaws in android apps is easy. In: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity. p. 21–22. SPLASH '13 (2013)