Checking Compliance of AADL Models with Modeling Guidelines using Resolint

Isaac Amundson
Collins Aerospace

Abstract

Certification standards for high-assurance systems include objectives for demonstrating compliance of process artifacts such as requirements and code with style guidelines and other standards. With the emergence of model-based development, similar objectives have been specified that apply to models. Demonstration of compliance is often achieved by employing a static analysis *linter* tool. This paper describes Resolint, an open-source, lightweight linter tool for checking compliance of Architecture Analysis and Design Language (AADL) models with modeling guidelines. AADL enables engineers to describe the key elements of distributed, real-time, embedded system architectures with a sufficiently rigorous semantics. In addition, AADL provides an annex mechanism for extending the base language, enabling new kinds of analyses and tool support. Resolint uses the AADL annex capability to provide a language for specifying style guide rule sets. It is implemented as a plugin for the Eclipse-based Open Source AADL Tool Environment (OSATE) and includes an engine for evaluating whether an AADL model complies with the specified rule sets. Results of the Resolint analysis are displayed to the user and can even be automatically incorporated as evidence in a system assurance case using the companion Resolute tool. To illustrate the features of Resolint, we present three use cases involving the assurance of embedded avionics applications. We further describe how we applied Resolint in the evaluation, synthesis, and assurance of a cyber-resilient UAV surveillance application developed on the DARPA Cyber Assured Systems Engineering (CASE) program.

Introduction

Certification standards for high-assurance systems include objectives for demonstrating compliance of process artifacts with design standards. For example, the RTCA DO-178C [1] guidance provides a means of compliance with airworthiness regulations for airborne software in commercial aircraft, and includes the following objective for software architecture:

> *6.3.3.e Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restriction and design construct rules.*

Similar objectives are specified that apply to high-level requirements (6.3.1.e), low-level requirements (6.3.2.e), and source code (6.3.4.d).

As interest in model-based development (MBD) of critical avionics software grew more prevalent, the RTCA DO-331 [2] guidance was introduced as a supplement to DO-178C. DO-331 provides clarification on the expected use of MBD technologies as well as additional considerations for ensuring safety and integrity goals are met. Specifically, the following objective is included:

> *MB.6.3.3.e Conformance to standards: When software architecture is expressed by a model, the objective is to ensure that the Software Model Standards were followed during the software design process and that deviations from the standards are justified.*

Some of the requirement, design, and modeling standards referred to by these objectives may be standardized by an industry body and widely used across that domain (e.g., MISRA C [3] and MAB[4] in the automotive industry). However, it is typically up to a development organization to select or author the standard, and then demonstrate compliance with it. Demonstration of compliance is associated with a review such that a design artifact and the design standards are inputs to the review (in addition to requirements, known anomalies, etc.), with the principal output being a review report containing stakeholder signatures of acceptance [5].

In general, design standards are comprised of a set of rules that govern appearance, naming conventions, techniques, structure, and other design constraints. Although demonstration of compliance with design standards can be achieved by manual review, in which the design is visually inspected and determined to be in compliance with each rule, typically automated static analysis checking tools (also referred to as *linter* tools) are deployed.

This paper describes Resolint, an open-source, lightweight linter tool for checking compliance of Architecture Analysis and Design Language (AADL) [6] models with modeling standards. Resolint is implemented as a plug-in for the Open Source AADL Tool Environment (OSATE), the AADL reference implementation developed by the Software Engineering Institute at Carnegie Mellon University.

The increasing complexity of safety-critical systems directly impacts the certification effort. Modeling and analyzing these systems enable early detection and removal of issues, thereby improving quality and reducing overall development and certification cost. To manage complexity for analysis, it is desirable to model the system hierarchically, starting with the system architecture and refining to increasing levels of detail.

AADL (SAE standard AS5506 [7]) enables engineers to describe the key elements of distributed, real-time, embedded system architectures. An AADL model includes the components in a system, their interfaces, properties, information flows, and interconnections.

Pre-print for the following paper published by the SAE:

Amundson, I., "Checking Compliance of AADL Models with Modeling Guidelines using Resolint," SAE Technical Paper 2023-01-0995, 2023, doi:10.4271/2023-01-0995.

Hardware components include devices, buses, memory, and processors, while software components include processes, threads, data, and subprograms. AADL also provides an annex mechanism for extending the base language, enabling new kinds of specification, analysis, and tool support.

In previous work, we developed Resolute [8], an AADL annex language and tool for specifying and instantiating assurance patterns and evaluating the resulting assurance arguments. Because safety-critical products generally undergo certification at the system level, there is a natural mapping between a system design and the corresponding assurance argument. Resolute takes advantage of this alignment by enabling the specification of the assurance argument directly in an AADL system model. The assurance case can then be instantiated and evaluated with elements specified in the model.

For checking compliance with modeling standards, Resolint rules are specified in the Resolute annex of AADL models. This is because Resolint rules use the same grammar as Resolute claims. In addition, the Resolute evaluation engine is used to determine whether the AADL model is in compliance with Resolint rules. Otherwise, Resolint and Resolute are two different tools with two different use cases. Future versions of Resolint may have greater independence from Resolute.

The remainder of this paper describes the Resolint tool and presents three use cases involving the assurance of embedded avionics applications. We further describe how we applied Resolint in the evaluation, synthesis, and assurance of a cyber-resilient UAV surveillance application developed on the DARPA Cyber Assured Systems Engineering (CASE) program.

Related Work

Static code analysis tools have been in use since the 1970s when the Lint tool for C programs was developed at Bell Labs by Stephen C. Johnson [9]. The utility of these code analysis tools was not lost on the software development community, and since then there have been numerous such tools developed for most software languages, compilers, and development environments [10]. One of the oldest, PC-lint [11], was first released in 1985 by Gimpel Software (recently acquired by Vector Informatik) and is still for sale today.

With the advent of languages and tools for model-based development, similar types of linter utilities were made available, typically as a feature of the modeling environment. The MathWorks released M-Lint (whose development was also led by Johnson) for MATLAB code, and later, Model Advisor and Simulink Check for Simulink models [12]. Similarly, MES Model Examiner [13] is a stand-alone static analyzer for Simulink models. Dassault Cameo Systems Modeler [14] and Ansys SCADE [15] modeling frameworks include linter utilities as well.

There are not many linter options for AADL. For commercially licensed software, Ellidiss Technologies developed AADL Inspector [16], which is a stand-alone tool that includes features for multiple types of analyses (static, timing, safety, security, etc.) of AADL models and several of its annexes. OSATE itself includes a syntax validation mechanism that could be adapted as a kind of linter, but adding or modifying rules (which are essentially hard-coded in the validator) would be difficult to accomplish for the average user. To the best of our knowledge, Resolint is the only open-source AADL linter tool that is a fully integrated plugin for OSATE.

Resolint

Rules derived from sources such as development standards, checklists, and modeling guidelines can be encoded in Resolint and embedded in the Resolute annex of an AADL model. Because rules in Resolint are represented using the same language as Resolute claims, a brief overview of Resolute is provided here.

Resolute

In Resolute, users formulate claims and logical rules for satisfying those claims, which Resolute uses to construct assurance cases. Both the claims and rules are parameterized by variables, which are instantiated using elements from the model. This connects the assurance case directly to the AADL model and means that changes to the model can result in changes to the assurance case. Resolute can then automatically evaluate the assurance argument by extracting supporting evidence directly from the model.

For example, the Resolute `goal` in Figure 1 specifies that a given process `p` is protected from alterations by other processes. When this goal is instantiated with a specific process component within an AADL model, Resolute can determine whether the claim is substantiated by evaluating the claim's Boolean expression (the nested `forall` statements starting on line 5 in this example).

```
1  goal memory_protection(p : process) <=
2      ** "Process " p " memory is protected from other
3          processes" **
4      strategy: "Argue over bound processes";
5      forall(mem : memory) . bound(p, mem) =>
6          forall(q : process) . bound(q, mem) =>
7          memory_safe_process(q)
```

Figure 1. Example Resolute claim.

In the example, all memory components that process `p` is bound to are evaluated (line 5), and if another process `q` is also bound to one of the memory components (line 6), `q` must have a property that identifies it as a memory-safe process (line 7), or the claim will fail. Here, `memory_safe_process()` is another user-defined claim that includes its own logical expression to determine whether a process is memory safe.

In addition to supporting first-order logic, the Resolute language includes a collection of common functions for traversing and evaluating AADL models. An abbreviated list of representative built-in functions is shown in Figure 2. In total, more than 70 built-in functions are defined.

- **union** (collection, collection) : collection
- **intersect** (collection, collection) : collection
- **length** (collection) : int
- **member** (element, collection) : bool
- **sum** (numeric collection) : numeric
- **upper_bound** (range) : numeric
- **lower_bound** (range) : numeric
- **property** (named element, property) : value
- **parent** (named element) : named element
- **name** (named element) : string
- **type** (named element) : Classifier
- **source** (connection) : connection endpoint
- **destination** (connection) : connection endpoint
- **features** (named element) : {feature}
- **subcomponents** (named element) : {component}
- **bound** (component, binding target) : bool

Figure 2. Partial list of built-in Resolute functions.

Not only does Resolute provide language constructs for traversing and evaluating an architecture model, but it also includes a plug-in mechanism that enables users to run external analysis tools. Resolute can then execute external analyses and use the analysis results to support assurance claims. A similar mechanism enables users to create Java function libraries that can be called from Resolute logic. Combined, these features support evidence generation and ingestion from artifacts both internal and external to the AADL workspace, enabling the assembly and evaluation of a comprehensive system assurance case.

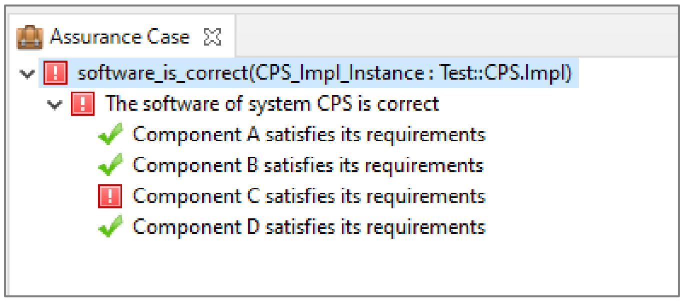
A few external analysis plug-ins are included with Resolute. For example, the `FileAccess` function library enables access to metadata and contents of file system artifacts. The `StringLib` function library provides an API to string manipulation functions, similar to those contained in the Java `String` class. Access to these types of function libraries aids in the specification of evidence evaluation rules that would not be possible using the base Resolute language and enables a broad collection of assurance evidence from sources outside the model.

For example, test results can be used to support an assurance claim that a given component implementation satisfies its requirements. In this scenario, suppose the test results were output by an automated test tool and reside in a file with a known format. To automatically determine if the assurance claim is supported, the contents of the test results file could be matched against a regular expression representing a passing case. In Resolute, such a claim, `component_satisfies_requirements()`, could be specified as shown starting on line 7 in Figure 3.

```
1   goal software_is_correct(sys : system) <=
2       ** "The software of system " sys " is correct" **
3       strategy: "Verify by testing";
4       forall(comp : subcomponents(sys)) .
5           component_satisfies_requirements(comp)
6
7   goal component_satisfies_requirements(comp : component) <=
8       ** "Component " comp " satisfies its requirements" **
9       let results : string =
10          FileAccess.getContents("results.log");
11      let pass_regex : string = name(comp) + "\\s*:\\s*PASSED";
12      StringLib.matches(results, pass_regex)
```

Figure 3. Resolute claim supported by external evidence.

Upon running Resolute on the system model, the generated assurance argument might look as shown in Figure 4. In this example, tests for components *A*, *B*, and *D* passed, whereas testing for component *C* had failures.


Assurance Case

```
software_is_correct(CPS_Impl_Instance : Test::CPS.Impl)
    The software of system CPS is correct
        Component A satisfies its requirements
        Component B satisfies its requirements
        Component C satisfies its requirements
        Component D satisfies its requirements
```

Figure 4. Assurance case generated by Resolute.

The Resolute User's Guide describes the full syntax for specifying claims, as well as the complete list of functions for querying structural properties of a model. It is included with Resolute or can be accessed at https://github.com/loonwerks/formal-methods-workbench/tree/master/documentation/resolute.

Formalizing Rules in Resolint

A Resolint rule is similar in structure to a Resolute claim. For example, an AADL modeling standard may contain the following rule:

> *Threads should have the Dispatch_Protocol property specified.*

In Resolint this would be represented as

```
dispatch_protocol_specified() <=
    ** "Threads should have the Dispatch_Protocol property
        specified" **
    forall(t : thread) .
        lint_check(t, has_property(t, Dispatch_Protocol))
```

Similarly, the rule

> *Threads can only specify a Dispatch_Protocol of Periodic or Sporadic.*

would be specified as

```
valid_dispatch_protocol() <=
    ** "Threads can only specify a Dispatch_Protocol property
        of Periodic or Sporadic" **
    forall(t : thread) .
        lint_check(t,
            has_property(t, Dispatch_Protocol) =>
            property(t, Dispatch_Protocol) = "Sporadic" or
            property(t, Dispatch_Protocol) = "Periodic")
```

The syntax of a rule is

```
<Rule> ::= <name> '(' (<Param> ',')* ')' '<='
'**' <rule_description> '**' <Expression>

<Param> ::= <param_name> ':' <Type>
```

where `<name>` is a string representing the rule name, `<rule_description>` is a textual description of the rule, `<param_name>` is a string representation of a parameter name, `<Type>` is a valid Resolute type, and `<Expression>` is a valid Resolute logical expression representing the rule.

Note that both of the above rules contain a call to `lint_check()`. `lint_check()` is a provided function that enables Resolint to capture the specific model element that violates the rule. The definition of `lint_check()` (shown in Figure 5) is specified in Resolint.aadl, which is included with the tool as an AADL plug-in contribution.

```
lint_check(element : aadl, result : bool) <=
    ** element **
    result
```

Figure 5. The `lint_check()` function for linking a rule violation with a model element.

The function takes an AADL element and a Boolean value. The Boolean value is the result of the rule check and becomes the result of the claim without modification. If its value is `false`, Resolint internally keeps track of the AADL element that violated the rule in order to provide the user with a direct reference in the Eclipse *Problems* pane.

Two other `lint_check` functions are provided: `lint_check_set()` and `lint_check_list()`. These are used when multiple elements are referenced in a rule. For example, the `one_process()` rule shown in Figure 6 will be violated if multiple process components exist that contain threads or thread groups. If this is the case, the user should be presented with the set of all such processes. In this example, `lint_check_set()` evaluates the size of the set of processes containing threads, and if the set is not empty, all processes in the set will be flagged.

```
one_process() <=
    ** "The model must contain at least one process bound to a
        processor" **
    let procs : {process} = {p for (p : process) |
        exists(pr : processor) . is_bound_to(p, pr)};
    lint_check_set(procs, size(procs) > 0)
```

Figure 6. Use of the `lint_check_set()` claim for linking a rule violation with multiple model elements.

The `lint_check()` functions are not necessary for Resolint to check rules and display results. However, they are currently necessary to hyperlink Resolint analysis results with the AADL elements that are violating the rules. Future versions of Resolint may eliminate the need for the `lint_check()` functions.

Creating Rulesets

Resolint rules can be grouped into *rulesets*. These are useful for organizing rules corresponding to different guidelines and standards, such as organizational styles, customer requirements, certification guidelines, and tool constraints. Rulesets also provide the ability to specify the severity of the rule violation; that is, the type of message the user should receive if the rule is found to be violated. Three levels of severity are supported. From least to most severe, they are *info*, *warning*, and *error*.

The syntax of a ruleset is

```
<Ruleset> ::= ‘ruleset’ <name> ‘{’ (
<Resolint_Statement> )* ‘}’

<Resolint_Statement> ::= (‘info’ | ‘warning’ |
‘error’) ‘(’ <Rule_Reference> ‘)’
```

where `<name>` is a string representing the ruleset name and `<Rule_Reference>` is the function name of a Resolute claim representing the rule.

Resolint statements are interpreted such that if the referenced rule evaluates to false, the user will receive a message marker of the severity indicated by the `info`, `warning`, or `error` keyword.

An example ruleset is depicted in Figure 7.

```
ruleset CASE_Tools {
    -- The model must contain at least one
    -- process bound to a processor
    error( one_process() )
    -- A seL4 process must have at most one
    -- thread subcomponent
    error( one_thread() )
    -- AADL modes are not currently supported
    -- by CASE tools
    warning( modes_ignored() )
    -- AADL flows are not currently supported
    -- by CASE tools
    warning( flows_ignored() )
    -- Threads should have the Dispatch_Protocol
    -- property specified
    warning( dispatch_protocol_specified() )
    -- Threads can only specify a Dispatch_Protocol
    -- property of periodic or sporadic
    error( valid_dispatch_protocol() )
    -- If a thread has a Dispatch_Protocol property
    -- value of Periodic then it must have valid Period
    -- and Compute_Execution_Time property values set
    warning( thread_periodic_protocol() )
    -- Integer types must be bounded
    error( bounded_integers() )
    -- Float types must be bounded
    error( bounded_floats() )
}
```

Figure 7. Example ruleset definition.

Checking Rules and Rulesets

In order to check that an AADL model complies with a set of rules, Resolint needs to know which rules or rulesets to check. This is specified using the `check` statement in an AADL component implementation. For example, the `check CASE_Tools` statement in the `MissionComputer.Impl` component in Figure 8 (line 22) will evaluate the `MissionComputer.Impl` system instance against the `CASE_Tools` ruleset.

```
system implementation MissionComputer.Impl
    subcomponents
        Radio: device Radio.Impl;
        UART: device UART.Impl;
        Proc: processor MC_Proc.Impl;
        Mem: memory MC_Mem.Impl;
        SW: process SW::SW.Impl;
    connections
        c1: port RadioRecv -> Radio.RecvIn;
        c2: port Radio.SendOut -> RadioSend;
        c3: port UartRecv -> UART.RecvIn;
        c4: port UART.SendOut -> UartSend;
    properties
        Actual_Processor_Binding =>
            (reference (Proc)) applies to SW;
        Actual_Memory_Binding =>
            (reference (Mem)) applies to SW;

    annex resolute {**
        -- Make sure this model complies with
        -- the CASE_Tools modeling standards
        check CASE_Tools
    **};
end MissionComputer.Impl;
```

Figure 8. Resolint `check` statement.

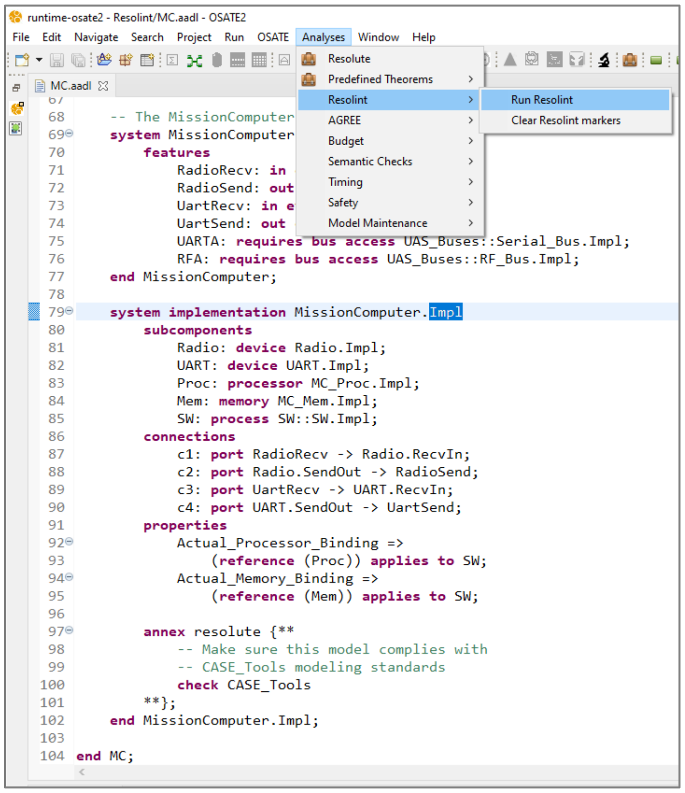
Similarly, individual rules that do not belong to a ruleset can also be checked, as in the example in Figure 9 (line 22). In this example, a single rule (`one_process()`) is being checked on `MissionComputer.Impl`, and if violated, an error will be issued.

```
1  system implementation MissionComputer.Impl
2      subcomponents
3          Radio: device Radio.Impl;
4          UART: device UART.Impl;
5          Proc: processor MC_Proc.Impl;
6          Mem: memory MC_Mem.Impl;
7          SW: process SW::SW.Impl;
8      connections
9          c1: port RadioRecv -> Radio.RecvIn;
10         c2: port Radio.SendOut -> RadioSend;
11         c3: port UartRecv -> UART.RecvIn;
12         c4: port UART.SendOut -> UartSend;
13     properties
14         Actual_Processor_Binding =>
15             (reference (Proc)) applies to SW;
16         Actual_Memory_Binding =>
17             (reference (Mem)) applies to SW;
18
19     annex resolute {**
20         -- The model must contain at least one
21         -- process bound to a processor
22         check error( one_process() )
23     **};
24 end MissionComputer.Impl;
```

Figure 9. Checking and specifying severity of individual rules.

Running Resolint

Resolint is run by selecting an AADL component implementation containing a `check` statement in its Resolute annex and running the tool from the Analyses menu included with OSATE (Analyses → Resolint → Run Resolint) as shown in Figure 10.

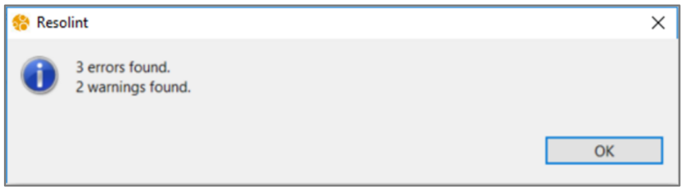


```
    -- The MissionComputer
    system MissionComputer
        features
            RadioRecv: in
            RadioSend: out
            UartRecv: in e
            UartSend: out
            UARTA: requires bus access UAS_Buses::Serial_Bus.Impl;
            RFA: requires bus access UAS_Buses::RF_Bus.Impl;
    end MissionComputer;

    system implementation MissionComputer.Impl
        subcomponents
            Radio: device Radio.Impl;
            UART: device UART.Impl;
            Proc: processor MC_Proc.Impl;
            Mem: memory MC_Mem.Impl;
            SW: process SW::SW.Impl;
        connections
            c1: port RadioRecv -> Radio.RecvIn;
            c2: port Radio.SendOut -> RadioSend;
            c3: port UartRecv -> UART.RecvIn;
            c4: port UART.SendOut -> UartSend;
        properties
            Actual_Processor_Binding =>
                (reference (Proc)) applies to SW;
            Actual_Memory_Binding =>
                (reference (Mem)) applies to SW;

        annex resolute {**
            -- Make sure this model complies with
            -- CASE_Tools modeling standards
            check CASE_Tools
        **};
    end MissionComputer.Impl;

end MC;
```

Figure 10. Running Resolint.

Resolint Output

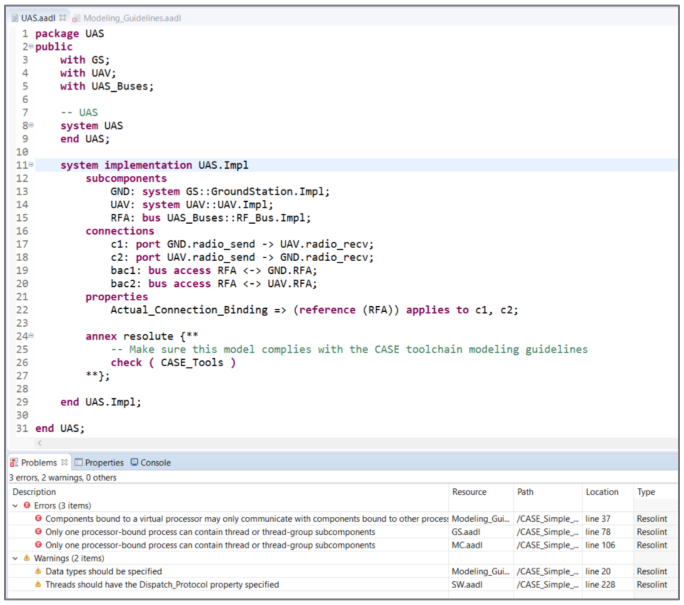
When the Resolint analysis is complete a message box will inform the user whether any rule violations were discovered, and if so, how many of each severity type (see Figure 11).



3 errors found.
2 warnings found.

Figure 11. Resolint result message.

In addition, the list of rule violations will appear in the standard Eclipse Problems pane, as shown in Figure 12.



UAS.aadl | Modeling_Guidelines.aadl

```
package UAS
public
    with GS;
    with UAV;
    with UAS_Buses;

    -- UAS
    system UAS
    end UAS;

    system implementation UAS.Impl
        subcomponents
            GND: system GS::GroundStation.Impl;
            UAV: system UAV::UAV.Impl;
            RFA: bus UAS_Buses::RF_Bus.Impl;
        connections
            c1: port GND.radio_send -> UAV.radio_recv;
            c2: port UAV.radio_send -> GND.radio_recv;
            bac1: bus access RFA <-> GND.RFA;
            bac2: bus access RFA <-> UAV.RFA;
        properties
            Actual_Connection_Binding => (reference (RFA)) applies to c1, c2;

        annex resolute {**
            -- Make sure this model complies with the CASE toolchain modeling guidelines
            check ( CASE_Tools )
        **};

    end UAS.Impl;

end UAS;
```

Problems | Properties | Console

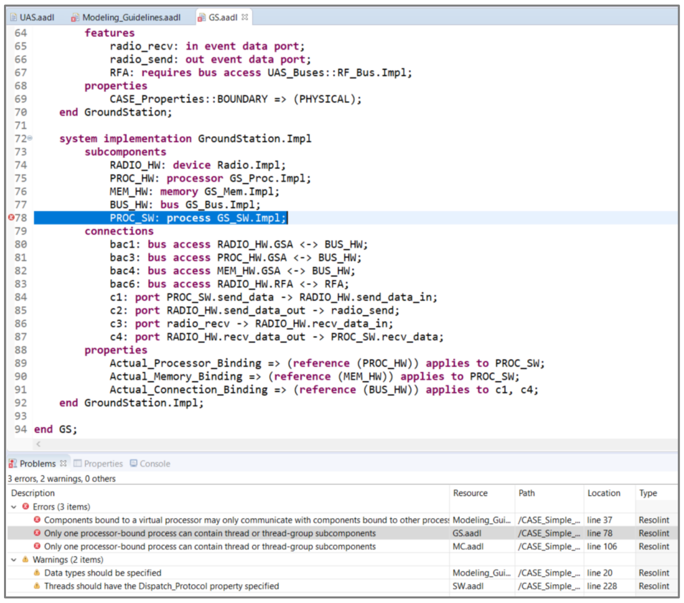
3 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (3 items)				
Components bound to a virtual processor may only communicate with components bound to other proces	Modeling_Gui...	/CASE_Simple_...	line 37	Resolint
Only one processor-bound process can contain thread or thread-group subcomponents	GS.aadl	/CASE_Simple_...	line 78	Resolint
Only one processor-bound process can contain thread or thread-group subcomponents	MC.aadl	/CASE_Simple_...	line 106	Resolint
Warnings (2 items)				
Data types should be specified	Modeling_Gui...	/CASE_Simple_...	line 20	Resolint
Threads should have the Dispatch_Protocol property specified	SW.aadl	/CASE_Simple_...	line 228	Resolint

Figure 12. Rule violations appear in the Eclipse *Problems* pane.

Double-clicking on an individual problem will open the package containing the AADL element violating the rule and highlight it, as well as place a marker with the corresponding severity in the margin, as shown in Figure 13.

Markers can be cleared by either fixing the rule violation and rerunning Resolint, or from the menu by selecting Analyses → Resolint → Clear Resolint markers.



UAS.aadl | Modeling_Guidelines.aadl | GS.aadl

```
        features
            radio_recv: in event data port;
            radio_send: out event data port;
            RFA: requires bus access UAS_Buses::RF_Bus.Impl;
        properties
            CASE_Properties::BOUNDARY => (PHYSICAL);
    end GroundStation;

    system implementation GroundStation.Impl
        subcomponents
            RADIO_HW: device Radio.Impl;
            PROC_HW: processor GS_Proc.Impl;
            MEM_HW: memory GS_Mem.Impl;
            BUS_HW: bus GS_Bus.Impl;
            PROC_SW: process GS_SW.Impl;
        connections
            bac1: bus access RADIO_HW.GSA <-> BUS_HW;
            bac3: bus access PROC_HW.GSA <-> BUS_HW;
            bac4: bus access MEM_HW.GSA <-> BUS_HW;
            bac6: bus access RADIO_HW.RFA <-> RFA;
            c1: port PROC_SW.send_data -> RADIO_HW.send_data_in;
            c2: port RADIO_HW.send_data_out -> radio_send;
            c3: port radio_recv -> RADIO_HW.recv_data_in;
            c4: port RADIO_HW.recv_data_out -> PROC_SW.recv_data;
        properties
            Actual_Processor_Binding => (reference (PROC_HW)) applies to PROC_SW;
            Actual_Memory_Binding => (reference (MEM_HW)) applies to PROC_SW;
            Actual_Connection_Binding => (reference (BUS_HW)) applies to c1, c4;
    end GroundStation.Impl;

end GS;
```

Problems | Properties | Console

3 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (3 items)				
Components bound to a virtual processor may only communicate with components bound to other proces	Modeling_Gui...	/CASE_Simple_...	line 37	Resolint
Only one processor-bound process can contain thread or thread-group subcomponents	GS.aadl	/CASE_Simple_...	line 78	Resolint
Only one processor-bound process can contain thread or thread-group subcomponents	MC.aadl	/CASE_Simple_...	line 106	Resolint
Warnings (2 items)				
Data types should be specified	Modeling_Gui...	/CASE_Simple_...	line 20	Resolint
Threads should have the Dispatch_Protocol property specified	SW.aadl	/CASE_Simple_...	line 228	Resolint

Figure 13. Automatic highlighting of model element responsible for rule violation.

Resolint Use Cases

In this section we present three use cases for Resolint and describe how we applied Resolint in the evaluation, synthesis, and assurance of a cyber-resilient UAV surveillance application developed on the DARPA CASE program.

The principal objective of the DARPA CASE program was the development of systems engineering tools that provide inherent cyber resiliency for complex cyber-physical systems. Our team developed the BriefCASE model-based systems engineering (MBSE) environment [17], which integrates formal design, verification, and code generation activities, while maintaining an assurance case comprised of proof artifacts and other evidence of correctness emitted by the tools. BriefCASE includes tools for:

1. Modeling cyber-physical system architectures in AADL
2. Analyzing AADL models and generating new requirements corresponding to discovered cyber vulnerabilities
3. Mitigating vulnerabilities through automated model transformations
4. Verifying model correctness using formal methods analysis
5. Synthesizing provably correct component and infrastructure code
6. Building to a formally verified separation kernel
7. Generating and viewing a cybersecurity assurance case

Resolint was developed on the CASE program and integrated with the BriefCASE environment. Through our work on the program developing and integrating CASE tools, as well as evaluating the resulting tool chain on real-world examples, we identified three key use cases that highlight the utility of Resolint.

Use Case 1: Enforce Appropriate Subset of AADL

Numerous AADL analysis tools have been developed since SAE standardization in 2004. However, not all tools are built to support the entire AADL syntax. Ideally these tools should be able to properly handle models containing incompatible syntax, but that is not always the case, and it may be desirable to create a syntax compatibility ruleset to check against the model in order to ensure compatibility before running the tool.

Because BriefCASE was comprised of multiple research tools (i.e., tools with low technology readiness level (TRL)) developed by different performers on the program, it became evident that a collection of exemplar models would be needed for testing tool functionality and integration. Since the tools were low TRL, especially early in the program, most did not provide support for the complete AADL language. By encoding the permissible (or conversely, non-permissible) syntax in tool-specific Resolint rulesets, it became very easy to create new exemplar models and immediately verify compatibility with a target tool.

For example, one of the CASE requirement generation tools was initially unable to recognize abstract features and feature groups, as well as thread group components. These constraints were encoded in Resolint (as shown in Figure 14) and checked on models before they were analyzed with the requirement generation tool. Note that a violation of the `no_thread_groups()` rule issues a warning rather than an error. This is because, although the requirement generation tool did not yet support thread group semantics, it was able to display an alert message to the user and continue its analysis.

```
ruleset Requirements_Tools {
    error( no_abstract_features() )
    error( no_feature_groups() )
    warning( no_thread_groups() )
}

no_abstract_features() <=
    ** "Abstract features are not supported" **
    forall(f : feature) . lint_check(f, is_abstract_feature(f))

no_feature_groups() <=
    ** "Feature groups are not supported" **
    forall(f : feature_group) . lint_check(f, false)

no_thread_groups() <=
    ** "Thread groups are not supported" **
    forall(t : thread_group) . lint_check(t, false)
```

Figure 14. Requirements generation tool ruleset for enforcing a subset of AADL.

Use Case 2: Ensure Appropriate Architecture Structure

Similar to enforcing acceptable AADL language elements, some tools require a specific structure of the architecture. For example, the SPLAT component synthesis tool [18] requires that target components have specific property associations as well as `guarantee` statements in an AGREE annex [19]. The HAMR build tool [20] requires that each process component contains only a single thread when building to an seL4 target [21], which is representative of the secure microkernel's notion of guaranteed time and space partitioning. Figure 15 shows excerpts of the code synthesis tools ruleset.

```
ruleset Synthesis_Tools {
    error( missing_guarantees() )
    error( one_thread_per_process() )
}

missing_guarantees() <=
    ** "Component is missing AGREE guarantee listed in
        Component_Spec" **
    forall(c : component) .
        has_property(c, Component_Spec) =>
        forall(spec : property(c, Component_Spec)) .
            lint_check(c, has_guarantee(c, spec))

one_thread_per_process() <=
    ** "A seL4 process must have at most one thread
        subcomponent" **
    let procs : {process} =
        {p for (p : process) |
            size(subcomponents(p)) > 1 and
            exists(pr : processor) .
                is_sel4_processor(pr) and
                processor_bound(p, pr)};
    lint_check_set(procs, size(procs) = 0)
```

Figure 15. Code synthesis tool ruleset for ensuring correct structure of AADL architectures.

Use Case 3: Check Compliance with Modeling Guidelines

On any high-assurance product development effort, rules derived from the above two use cases, along with additional style-based rules developed internally or perhaps originating from the customer, will be compiled into a collection of guidelines against which the model must be checked for compliance.

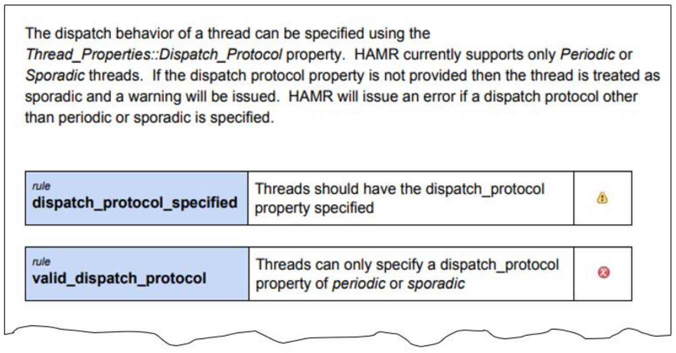
Style-based rules govern the format of the model rather than technical content and are in place to ensure consistency and other desirable quality attributes when multiple engineers work on the same project. Examples of formatting rules could include restricting model element names from containing underscore characters or requiring an `@author` tag in the comment block at the top of each file. For example, a development organization style rule for restricting underscore characters in component names could look as shown in Figure 16.

```
1   ruleset Dev_Org_Style {
2       warning( no_underscores_in_component_names() )
3   }
4
5   no_underscores_in_component_names() <=
6       ** "Underscore characters are not permitted in
7           component names" **
8       let comps : {component} =
9           {c for (c : component) |
10              StringLib.contains(name(c), "_")};
11      lint_check_set(comps, size(comps) > 0)
```

Figure 16. Development organization style rule for component naming.

Here, the name of each component in the model is retrieved and the `StringLib` function library is used to determine if the name contains an underscore (line 10). If this is the case, the offending component is flagged via the `lint_check` mechanism (line 11), and a warning is displayed to the user.

A modeling standards document was developed for the BriefCASE environment (an excerpt is shown in Figure 17) and the rules subsequently encoded in Resolint.


The dispatch behavior of a thread can be specified using the *Thread_Properties::Dispatch_Protocol* property. HAMR currently supports only *Periodic* or *Sporadic* threads. If the dispatch protocol property is not provided then the thread is treated as sporadic and a warning will be issued. HAMR will issue an error if a dispatch protocol other than periodic or sporadic is specified.

rule **dispatch_protocol_specified**	Threads should have the dispatch_protocol property specified	⚠
rule **valid_dispatch_protocol**	Threads can only specify a dispatch_protocol property of *periodic* or *sporadic*	⊗

Figure 17. An excerpt from the BriefCASE Modeling Guidelines.

BriefCASE includes a Resolute system cybersecurity assurance pattern that is instantiated with evidence generated by using the tool chain. A fragment of the assurance case in Goal Structuring Notation (GSN) [22] is shown in Figure 18. It contains a claim (goal) that the model has been checked against (and is in compliance with) the BriefCASE modeling guidelines. The corresponding Resolute goal is shown in Figure 19 (starting on line 8). When evaluating the assurance case, Resolute automatically runs Resolint via the built-in `resolint()` function (line 12) and includes the result as evidence in the evaluated assurance case.

The referenced assurance pattern and modeling guidelines are included with BriefCASE, which is open-source and can be downloaded from the BriefCASE project repository at https://github.com/loonwerks/BriefCASE.

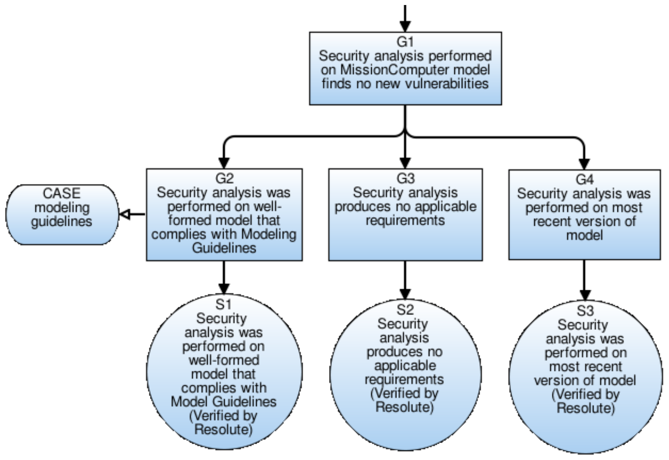

G1
Security analysis performed on MissionComputer model finds no new vulnerabilities
CASE modeling guidelines
G2
Security analysis was performed on well-formed model that complies with Modeling Guidelines
G3
Security analysis produces no applicable requirements
G4
Security analysis was performed on most recent version of model
S1
Security analysis was performed on well-formed model that complies with Model Guidelines (Verified by Resolute)
S2
Security analysis produces no applicable requirements (Verified by Resolute)
S3
Security analysis was performed on most recent version of model (Verified by Resolute)

Figure 18. GSN CASE assurance fragment. Goal `G2` argues that a security analysis was performed on a model that complies with the CASE modeling guidelines. Solution `S1` substantiates the claim using results from Resolint.

```
goal security_analysis_performed(sys : system) <=
    ** "Security analysis performed on " sys " model finds
        no new vulnerabilities" **
    model_complies_with_guidelines(sys) and
    analysis_performed_on_current_model() and
    analysis_produces_no_applicable_requirements()

goal model_complies_with_guidelines(sys : system) <=
    ** "Security analysis performed on well-formed model
        that complies with Model Guidelines" **
    context: "CASE modeling guidelines";
    resolint(sys)

goal analysis_performed_on_current_model() <=
    ** "Security analysis was performed on the most recent
        version of the model" **
    BriefCASE.analysis_performed_on_current_model()

goal analysis_produces_no_applicable_requirements() <=
    ** "Security analysis produces no new applicable
        requirements" **
    BriefCASE.analysis_produces_no_applicable_requirements()
```

Figure 19. Resolute implementation of CASE assurance pattern.

Conclusion

This paper describes the Resolint tool for checking compliance of AADL models with modeling standards. With the emergence of high-assurance certification objectives targeting aspects of model-based development, it is important to have access to the proper tools and methods for satisfying those objectives. Resolint addresses this need by providing a means for generating compliance evidence that can be used to support assurance claims.

Because the result of Resolint can be used as evidence for satisfying certification objectives, qualification of the tool will be required in most certification domains (for example, see RTCA DO-330 [23]). Qualification activities are typically performed on a per-product basis. Some tools provide qualification kits to help development organizations reduce the certification costs associated with using the tool. Although Resolint does not include a qualification kit at present, this could be considered for future releases.

We are currently in the process of implementing a headless version of Resolint that can be run from a command line for incorporation into a CI/CD process. This will enable models to automatically be checked for compliance with modeling guidelines when they are checked into a repository.

Looking ahead, we hope to improve Resolint by exploring GUI enhancements, implementing a simplified mechanism for linking rule violations with model elements, and potentially even decoupling from Resolute entirely. We would also like to provide better support for distinguishing between an AADL system instance and the individual AADL packages that comprise a project. Currently, the evaluation of Resolint rules is performed against an *instantiated* AADL system model. Although built-in Resolute functions are provided for navigating elements within an AADL package, it may not be intuitive to the user how best to do this. Finally, we hope to continue keeping Resolint up to date with the latest AADL and OSATE versions as they are released.

Resolint is open-source software through a BSD-3 license. It is currently packaged with Resolute, which can be installed in OSATE from the Help → Install Additional OSATE Components menu or downloaded from the Resolute project repository at https://github.com/loonwerks/Resolute.

References

1. RTCA DO-178C, "Software considerations in airborne systems and equipment certification," 2011.
2. RTCA DO-331, "Model-Based Development and Verification Supplement to DO-178C and DO-278A," 2011.
3. MISRA Consortium, "MISRA C:2012 Guidelines for the use of the C language in critical systems," 2012.
4. The MathWorks Advisory Board, "MAB Guidelines," Accessed November 6, 2022. https://www.mathworks.com/solutions/mab-guidelines.html
5. IEEE, "IEEE Standard 1028-2008 for Software Reviews and Audits," 2008.
6. Feiler, P. H. and Gluch, D. P., "Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language," 1st ed. Addison-Wesley Professional, 2012.
7. SAE AS5506D, "Architecture Analysis and Design Language (AADL)," April 22, 2022.
8. Gacek, A., Backes, J., Cofer, D., Slind, K., et. al. "Resolute: an assurance case language for architecture models," Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014, Michael Feldman and S. Tucker Taft (Eds.). ACM, 19–28.
9. Johnson, S. C., "Lint, a C program checker," Murray Hill: Bell Telephone Laboratories, 1977.
10. Wikipedia. "List of tools for static code analysis". Accessed November 6, 2022. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
11. Vector Informatik. "PC-lint Plus". Accessed November 6, 2022. https://pclintplus.com/pc-lint-plus/
12. The MathWorks. "Simulink Check". Accessed November 6, 2022. https://www.mathworks.com/products/simulink-check.html
13. MES. "Model Examiner". Accessed November 6, 2022. https://model-engineers.com/en/quality-tools/mxam/
14. Dassault Systemes. "Cameo Systems Modeler". Accessed November 6, 2022. https://www.3ds.com/products-services/catia/products/no-magic/cameo-systems-modeler/
15. Ansys. "SCADE". Accessed November 6, 2022. https://www.ansys.com/products/embedded-software/ansys-scade-suite

16. Ellidiss Technologies. “AADL Inspector”. Accessed November 6, 2022. https://www.ellidiss.com/products/aadl-inspector/
17. Cofer, D., Amundson, I., Babar, J., Hardin, D., et. al. “Cyber Assured Systems Engineering at Scale,” IEEE Security and Privacy, May-June 2022.
18. Mercer, E., Slind, K., Amundson, I., Cofer, D., et. al., “Synthesizing verified components for cyber assured systems engineering,” 24th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021). October 2021.
19. Cofer, D., Gacek, A., Miller, S., Whalen, M., et. al. “Compositional verification of architectural models”. NASA Formal Methods, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg. 2012. pp. 126–140.
20. Hatcliff, J., Belt, J., Robby, and Carpenter, T., “HAMR: An AADL multi-platform code generation toolset,” 10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), ser. LNCS, vol. 13036. 2021. pp. 274–295.
21. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., et. al. “seL4: formal verification of an OS kernel,” Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP 2009). Big Sky, Montana, USA., October 11-14, 2009. J. N. Matthews and T. E. Anderson, Eds. ACM, 2009. pp. 207–220.
22. The Assurance Case Working Group SCSC-141B. “Goal Structuring Notation Community Standard (Version 2),” 2011.
23. RTCA DO-330, “Tool Qualification Supplement to DO-178C and DO-278A,” 2011.

Contact Information

Isaac Amundson is a research engineer at Collins Aerospace. He can be reached at isaac.amundson@collins.com.

Acknowledgments

We would like to thank John Shackleton (Galois) and Jason Belt (Kansas State University) for authoring the bulk of the CASE modeling guidelines referred to in this paper. The guidelines provided a diverse assortment of rules covering much of the AADL grammar, thus enabling development of a more robust tool. In addition, we thank all the CASE performers that put Resolint through its paces throughout the program. Finally, thanks to the reviewers for providing constructive feedback on this paper.

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