

Collins BriefCASE User's Guide

June 2021

BriefCASE: Build and Research Interface Environment for Cyber Assured Systems Engineering



Contents

Introduction	3
1. Generating and Importing Cyber Requirements.....	3
Invoking the Cyber Requirements Tool	3
Importing requirements from a file	4
Requirements Manager	6
Modifying Imported Requirements	7
Notes	8
2. Model Annotations	9
CASE Properties.....	10
Property Association User Interface	10
3. Model Transformations	11
Filter	11
CASE Filter Properties	14
Filter Synthesis	15
Design Assurance	15
Attestation	16
Design Assurance	20
Virtualization.....	21
Design Assurance	25
Monitor	25

Design Assurance	32
seL4	34
Design Assurance	38
4. SPLAT	39
Supported Constraints and Limitations	40
Example	41
5. HAMR	42
HAMR Translation Architecture	44
Examples	45
6. BriefCASE-Compatible Tools	46
AGREE	46
Resolute	46
Resolint	47
7. AADL Modeling Guidelines	47

Introduction

The Collins CASE team is developing tools to assist system engineers to design cyber-physical systems that must satisfy cyber-resiliency requirements. Our tools are based on AADL system models and support the import of cyber requirements, formal verification of requirements, system transformations to incorporate cyber-resilient design patterns, and building high-assurance implementations from the verified models.

1. Generating and Importing Cyber Requirements

BriefCASE interfaces with tools to generate cyber-security requirements for AADL models to improve their cyber-resiliency. In this section, we describe how to:

1. Run a cyber requirements tool from within the BriefCASE environment
2. Import the generated cyber-security requirements into the AADL model

There are two cyber-requirements tools that are part of the CASE program: GearCASE and DCRYPPS. These tools can be executed from the BriefCASE menu on the menu. On execution, they generate a file containing cyber-security requirements, which can then be imported into the AADL model. This workflow is illustrated below with respect to the GearCASE tool. The same steps can be taken when using DCRYPPS. Note that currently GearCASE and DCRYPPS are not packaged with the Collins BriefCASE environment, and will need to be installed separately. Future versions of BriefCASE may include them.

Invoking the Cyber Requirements Tool

To analyze a model for cyber vulnerabilities, an AADL file containing the top-level system implementation in the model must be open and a top-level system implementation selected. A cyber requirements tool can then be executed from the menu. For example, to run GearCASE, select the AADL component implementation to analyze in the Outline pane (or in the text editor), then choose BriefCASE → Cyber Requirements → Generate Requirements → GearCASE from the main menu, as shown in Figure 1.

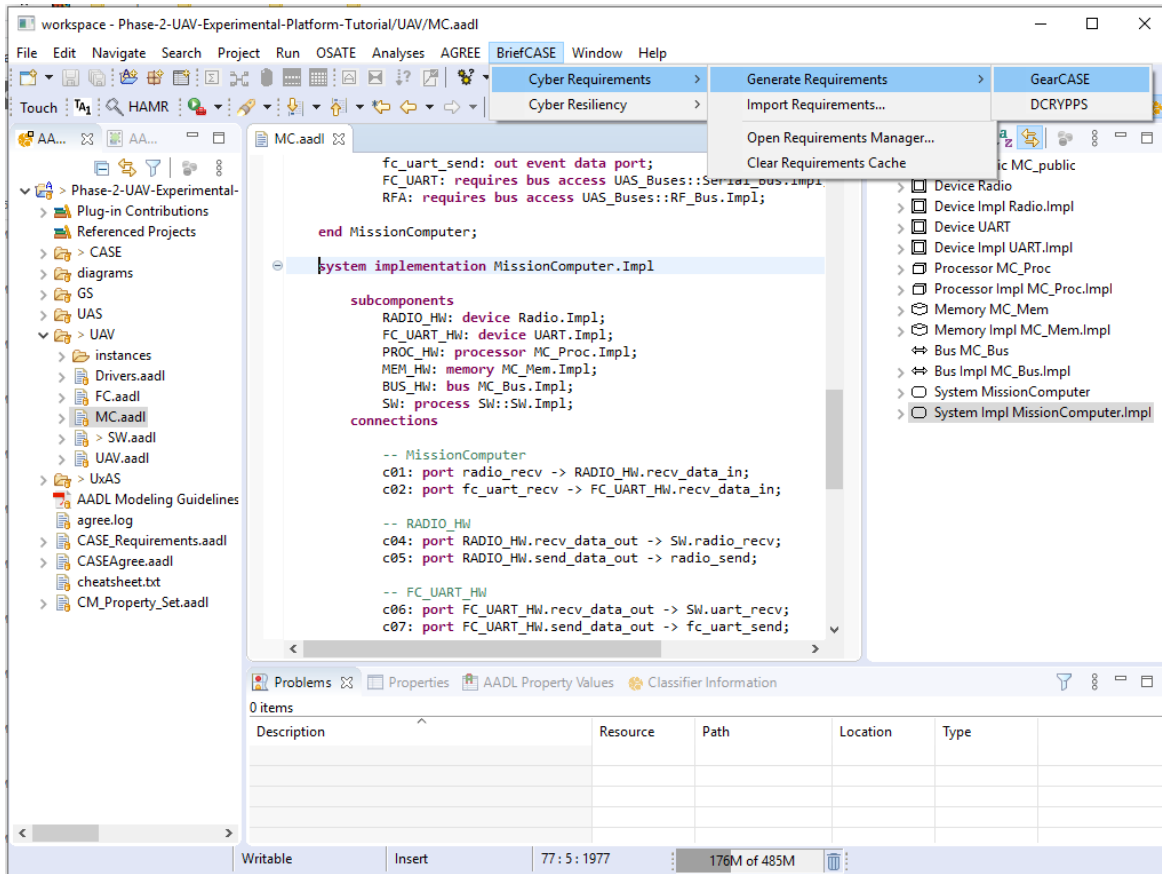


Figure 1. Invoking the GearCASE cyber requirements tool.

Importing requirements from a file

When the cyber requirements tool completes its analysis, it will output a requirements file to the Requirements folder in the project directory. To view the generated requirements, select the BriefCASE → Cyber Requirements → Import Requirements... menu item, as shown in Figure 2.

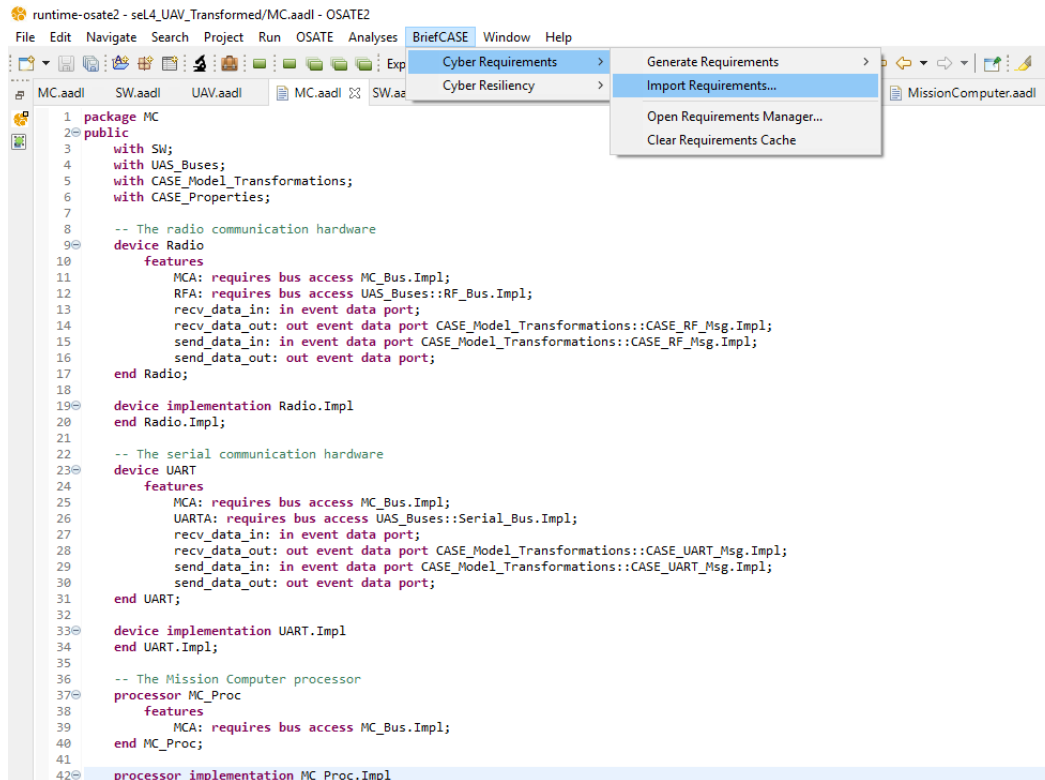


Figure 2. Launching the cyber requirements import wizard.

This brings up a file dialog to select the requirements file that was generated by the cyber requirements tool (see Figure 3). Selecting a valid requirements file will open the Requirements Manager and display the requirements.

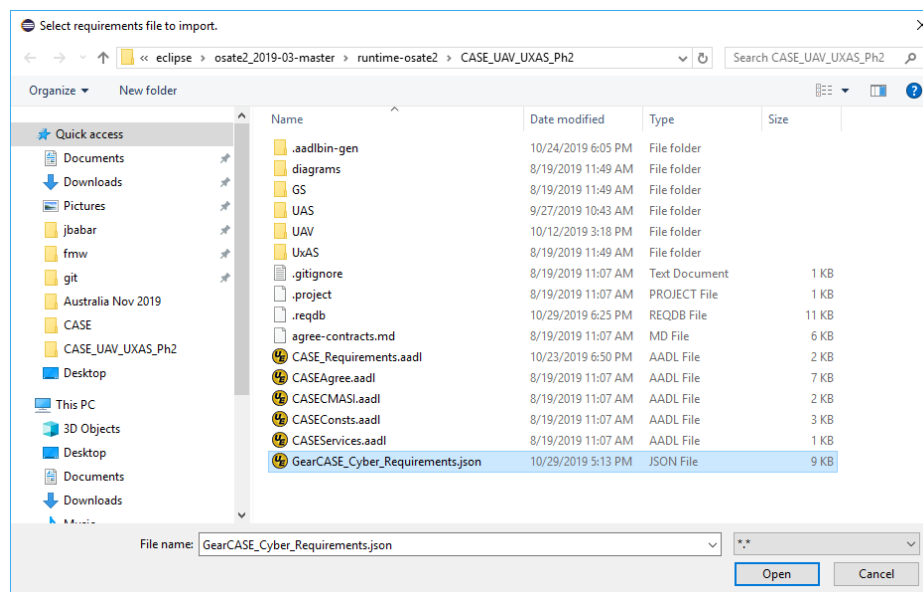


Figure 3. Requirements file selection dialog.

Requirements Manager

The requirements browser (left pane) shows the list of generated requirements as well as any requirements already present in the AADL model (see Figure 4). Selecting a requirement in the requirements browser shows additional details in the requirements viewer (right pane).

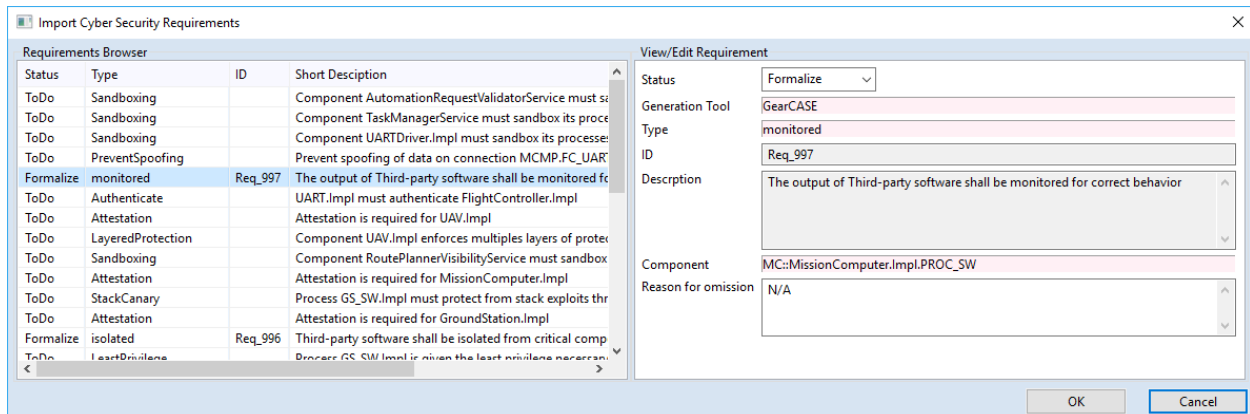
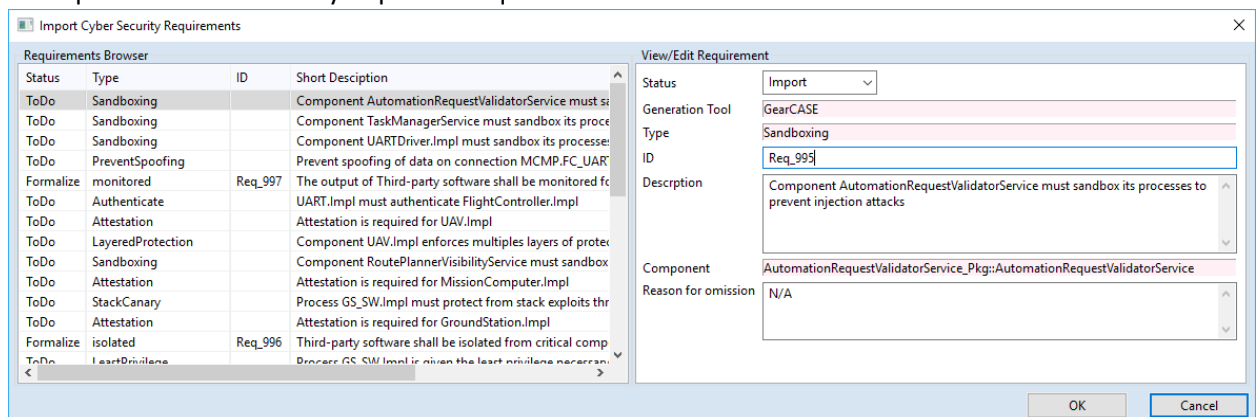


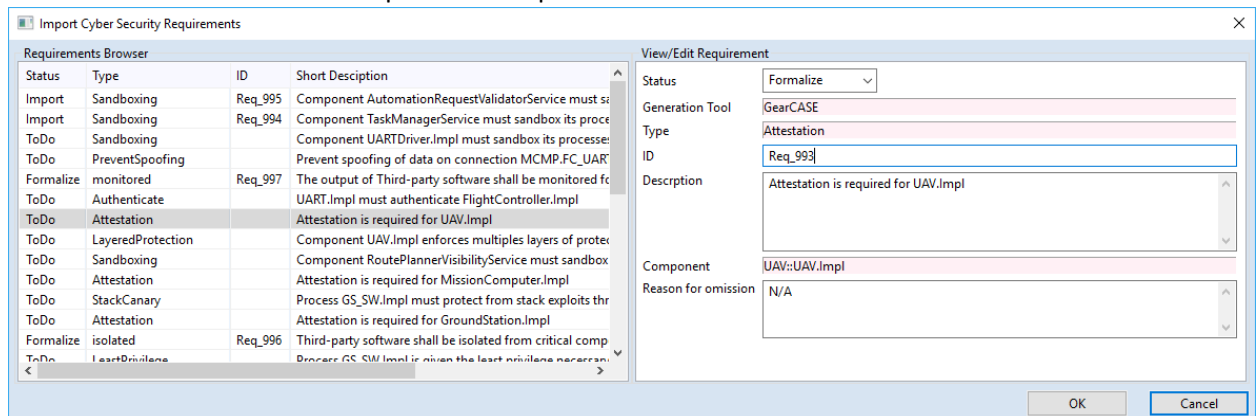
Figure 4. Requirements manager.

Through the Requirements Manager interface, the user can perform the following tasks:

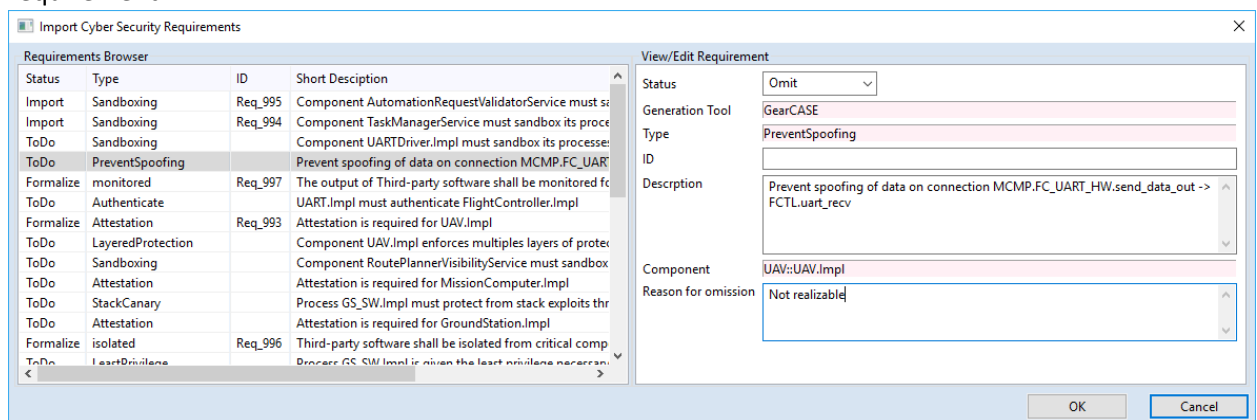
1. **Import Requirements:** Adds skeletal Resolute claims to the AADL model. The user must provide a unique identifier for every imported requirement.



2. **Formalize Requirements:** Imports the requirement and adds skeletal Resolute claims, as well as a skeletal AGREE clause to the specified component in the AADL model.



3. **Omit Requirements:** Marks a requirement as omitted. If the requirement had already been imported into the AADL model, it is removed. The user should provide rationale for omitting the requirement.



4. **ToDo:** Marks a requirement as “to be processed”. Every newly generated requirement is marked “ToDo”.
5. Changing the status of an imported or formalized requirement to “Omit” or “ToDo” removes the requirement from the model.
6. Changes to the underlying AADL model are made only when the “OK” button is pressed.

Modifying Imported Requirements

Imported requirements can be modified in the Requirements Manager, invoked via BriefCASE → Cyber Requirements → Open Requirements Manager... (see Figure 5). This allows the user to import, omit, or modify requirements in the model over multiple sessions. It uses the same interface as the Import Requirements dialog, but populates it with cyber security requirements in the AADL model and any requirements marked *Omit* or *ToDo* that were generated by the most recent execution of the cyber requirements tool.

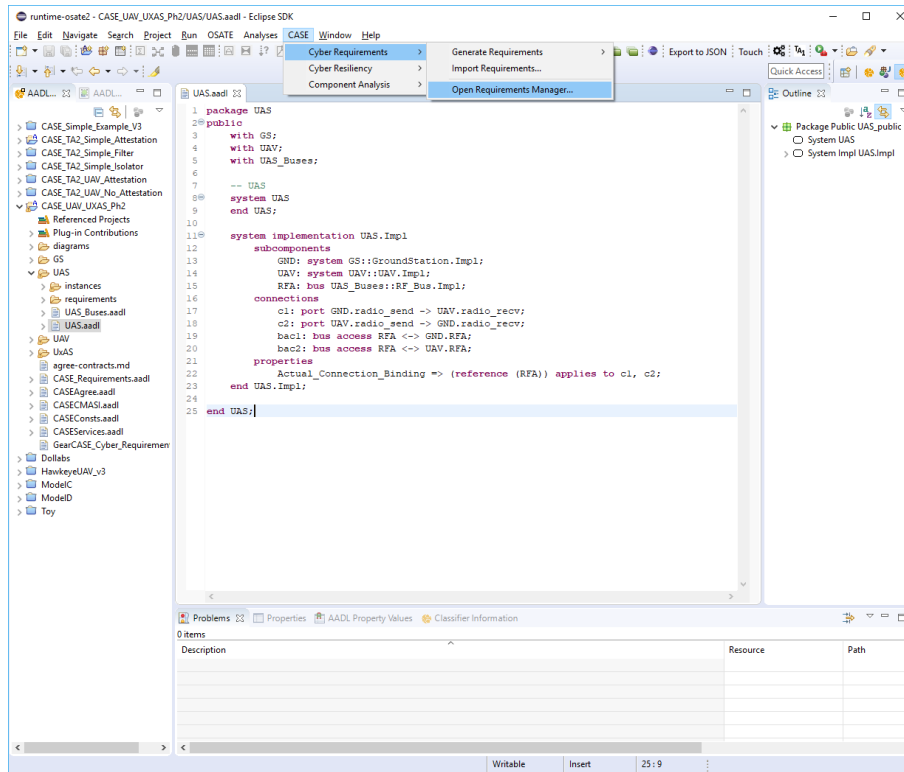
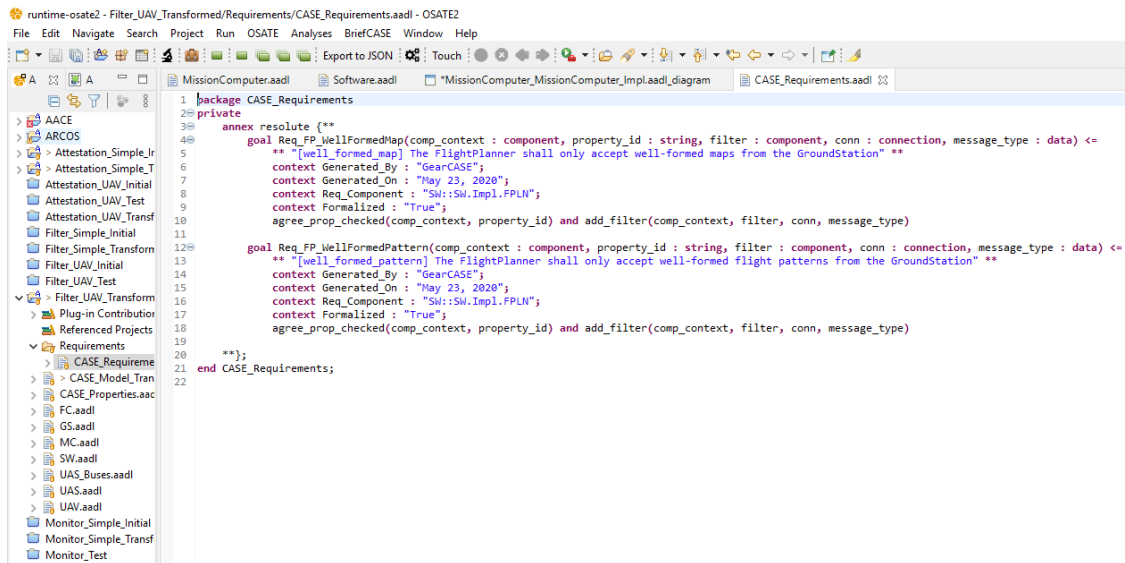


Figure 5. Opening the Requirements Manager.

Once the user has updated the AADL model to address imported cyber-security requirements, the cyber requirements tools can be called again to analyze whether new vulnerabilities were introduced in the modified model, generating new cyber security requirements. This iterative process continues until the user is satisfied that the AADL model is sufficiently cyber-resilient.

Notes

1. The file CASE_Requirements.aadl is added to the Requirements folder at the root level of the AADL project. It contains all the Resolute claim definitions added to the model.



2. The Collins TA2/5 tool maintains a list of requirements in the model. This permits editing of requirements across multiple BriefCASE sessions.
3. Files containing requirements to be imported must follow the prescribed JSON format.
4. When a new requirements file is imported (either directly or via the invocation of GearCASE or DCRYPPS), all previous requirements marked “ToDo” and “Omit” are removed from the internal requirements database, since these might not be relevant to the current state of the AADL model.
5. For maintaining traceability of requirements across multiple iterations of the TA1-TA2 tools, it is assumed that a snapshot of the requirements database is taken before every invocation of a cyber requirements tool.
6. Once a requirement has been imported, only its status can be changed. The sole exception is when a requirement is marked “Omit”, which would require editing the reason for its omission.
7. Every requirement has a *context*, which refers to an element (such as a connection or component) in the AADL model.
8. A requirement whose context is an AADL connection cannot be marked “Formalize”.

2. Model Annotations

AADL has been engineered as a general-purpose system architecture modeling language for embedded systems. As a result, the language specification does not natively include all elements necessary for capturing specific system properties, especially with respect to cyber-security. AADL does provide the ability to add custom properties to components. Annotating the model with CASE-specific properties will enable a more rigorous analysis. BriefCASE includes an AADL CASE property set, in addition to a user interface for simplifying the annotation of AADL components.

CASE Properties

The *CASE_Properties* AADL property set is provided with BriefCASE, and is accessible in an AADL project in the Plugin Contributions folder, as shown in the OSATE AADL Navigation pane in Figure 6.

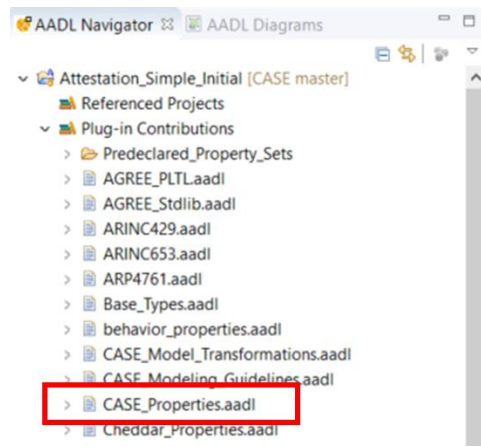


Figure 6. *CASE_Properties* property set

Property Association User Interface

To annotate an AADL element with a specific CASE property, select the element in the text editor and click the BriefCASE → Cyber Resiliency → Model Annotations... menu option. A window will open, similar to that pictured in Figure 7. Because different properties apply to different types of AADL elements, only those properties that apply to the selected element will be displayed in the Model Annotations window.

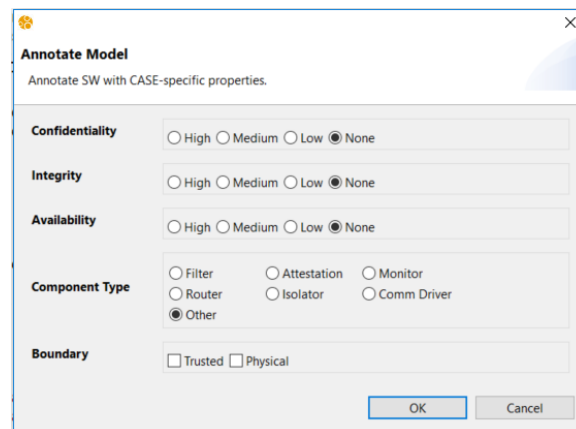


Figure 7. *Model Annotations*.

CASE properties that are already set for the selected component will be reflected in the Model Annotations window. When the appropriate properties are set, clicking OK will update the model.

Note that currently not all CASE properties may be visible in the Model Annotations window. Some properties may need to be manually entered in the text editor. This feature, along with the *CASE_Properties* AADL property set is currently undergoing revision, and may not function properly in some situations.

3. Model Transformations

BriefCASE provides a library of model transformations. However, the transformations may only be applied to specific AADL elements. Applying the transformations to AADL models that do not comply with these guidelines may have unintended consequences.

Filter

To illustrate the Filter transform, we use a simple producer-consumer example model. Both an initial model and a transformed model can be found here:

https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Filter/Simple_Example

Two AADL packages are included:

- Producer_Consumer.aadl – This is the initial model.
- CASE_Requirements.aadl – This is the package containing the requirement (in the form of a Resolute claim) that drives the filter transform.

A CASE Filter is added to a component's input port to ensure that only data that matches a specified regular expression arrives on that input port. To add a filter to the model, a connection must be selected that terminates at the input port of a component. For example, Figure 8 shows a thread subcomponent connected to its parent by connection c0. Also in the figure, connection c1 connects two thread components. A filter can be inserted onto either of these connections. However, a filter cannot be inserted onto connection c2, which connects the component to its parent. This is because a filter is always associated with the *input* port of a component.

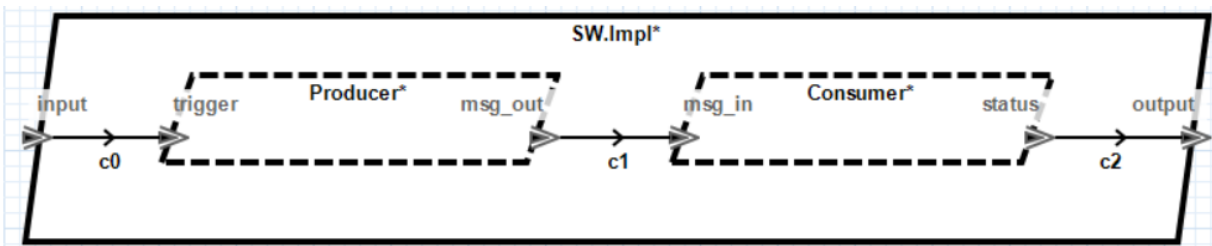


Figure 8. Initial model.

A filter can be added to the following AADL components:

- Thread
- Thread Group
- Process containing a single thread

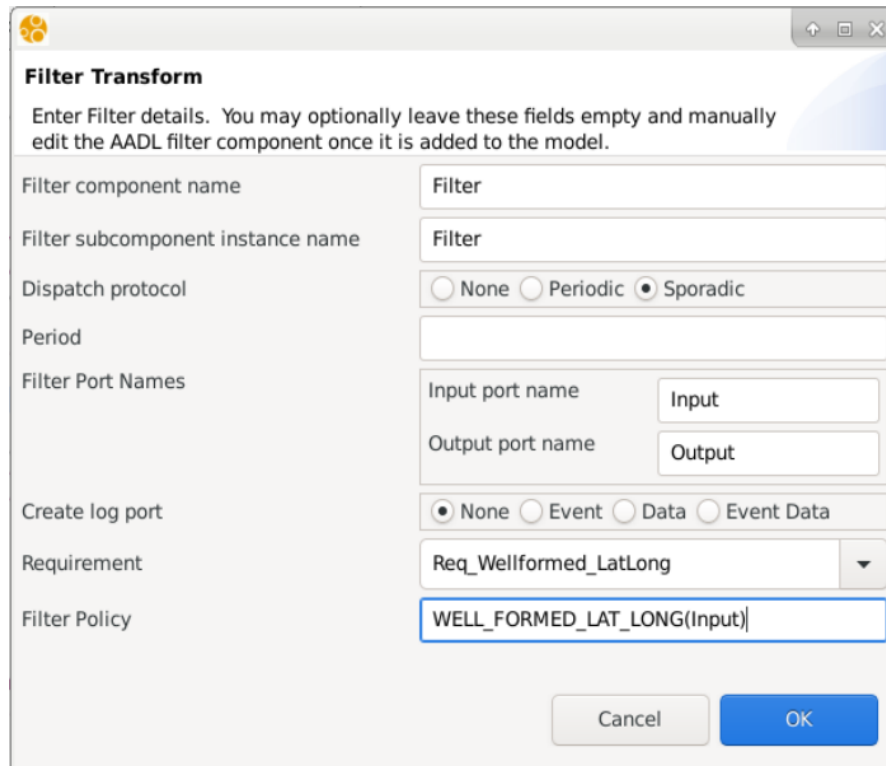
The model transform will insert a filter component that has the same component category as the target component it connects to, with two exceptions:

1. If the destination component is a thread group, the filter will be a thread.

2. If the destination component is a process containing a single thread, the filter will also be a process containing a single thread.

The latter supports the seL4 representation of components, in which each thread runs in its own address space. The transformation will also give the filter the same port type and category as the target component. Note that for System Build, the filter must be a software component (either a thread or process containing a single thread).

To insert a filter, select the connection in a component implementation that terminates at the component that requires filtered input (for example, in `Filter_Simple_Initial/Producer_Consumer.aadl`, select the `c1` connection on line 61). Note that currently the transformation can only be applied from within the OSATE text editor (future versions will enable applying the transformation from within the graphical editor). In the main menu, click the BriefCASE → Cyber Resiliency → Model Transformations → Add Filter... menu item. A wizard will open, as shown in Figure 9. The wizard enables the user to customize the filter, including providing the filter specification.



Filter Transform

Enter Filter details. You may optionally leave these fields empty and manually edit the AADL filter component once it is added to the model.

Filter component name:

Filter subcomponent instance name:

Dispatch protocol: ☐ None ☐ Periodic ☒ Sporadic

Period:

Filter Port Names:

Input port name:

Output port name:

Create log port: ☒ None ☐ Event ☐ Data ☐ Event Data

Requirement:

Filter Policy:

Figure 9. Add Filter wizard

The Filter transform will create an AADL component type and implementation, and insert them into the model. The component type name can be specified in the wizard (default is 'Filter'). The transform will then instantiate the filter as a subcomponent in the implementation containing the selected connection. The user may provide a name for the filter subcomponent, or use the default ('Filter'). If the field is left blank, the default name will be used. Note that if the specified name already exists, a number will be appended to the name to make it unique within the containing component implementation.

If the Filter is a thread component, the user can specify the Dispatch_Protocol and Period properties.

The user can provide descriptive names for the Filter input and output ports. Default names are 'Input' and 'Output', respectively.

By default, the CASE filter will drop any messages that do not satisfy the filter policy and no record of the malformed message will be retained. If the user wishes to log the event, an additional log port can be added to the filter. The user will need to specify the AADL port type (Event, Data, or EventData) and it will be up to the user to connect the log port to an appropriate "logger" component and implement logging functionality.

The requirement drop-down box lists all of the cyber-requirements that have been imported from Cyber Requirements tools. By specifying the cyber requirement that drives the filter transformation, the appropriate assurance argument can be constructed for demonstrating the requirement was addressed correctly. A requirement does not need to be selected to insert the filter, but it is highly recommended for construction of the proper system assurance case.

Finally, the user may provide the formal filter policy as an AGREE *expression*. This is typically done by referring to the incoming message on the Filter's input port. The message type will be the same as the source component's output port. Within the AGREE expression, refer to the Filter input port name by the name specified in the wizard. For example, if the message type is a signed integer, the Filter input port name is 'Input' and the filter should drop any message with a value less than zero, the AGREE statement will be:

```
Input >= 0;
```

In the producer-consumer example, we want to make sure a coordinate's latitude and longitude values are within the appropriate range. The filter expression is formalized in AGREE by the function `WELL_FORMED_LAT_LONG()` (Producer_Consumer.aadl, line 97), and so the filter property will simply be:

```
WELL_FORMED_LAT_LONG(Input);
```

Note that no syntax validation is performed on the AGREE expression. If it is malformed, it may not be imported into the model properly.

Clicking the OK button on the wizard will insert the Filter into the model, as shown in Figure 10 and Figure 11. The graphical representation is shown in Figure 12.

```

46⊖  thread Filter
47      features
48          Input: in event data port Coordinate.Impl;
49          Output: out event data port Coordinate.Impl;
50      properties
51          CASE_Properties::Filtering => 100;
52          CASE_Properties::Component_Spec => ("Filter_Output");
53⊖  annex agree {**
54⊖      guarantee Filter_Output "The filter output shall be well-formed" :
55⊖          if event(Input) and WELL_FORMED_LAT_LONG(Input) then
56              event(Output) and Output = Input
57          else
58              not event(Output);
59      **};
60  end Filter;
61
62⊖  thread implementation Filter.Impl
63      properties
64          Dispatch_Protocol => Sporadic;
65  end Filter.Impl;

```

Figure 10. Line 47: CASE_Filter component type; Line 63: CASE_Filter component implementation.

```

76⊖  process implementation SW.Impl
77      subcomponents
78          Producer: thread Producer.Impl;
79          Consumer: thread Consumer.Impl;
80          Filter: thread Filter.Impl;
81      connections
82          c0: port input -> Producer.trigger;
83          c1: port Producer.msg_out -> Filter.Input;
84          c2: port Consumer.status -> output;
85          c3: port Filter.Output -> Consumer.msg_in;
86⊖  annex resolute {**
87      prove Req_Wellformed_LatLong(this.Consumer, "Req_Wellformed_LatLong", this.Filter, this.c3, Coordinate.Impl)
88      **};
89  end SW.Impl;

```

Figure 11. Line 80: filter subcomponent; Lines 83,85: filter connections; Line 87: updated assurance claim call.

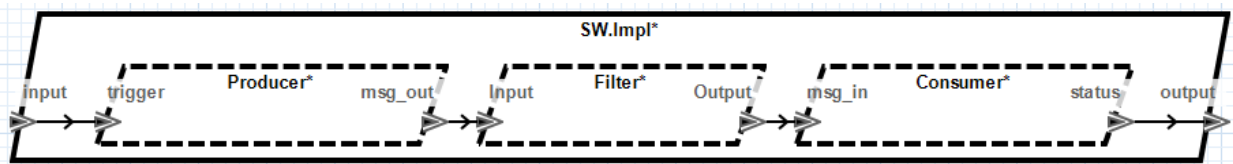


Figure 12. Transformed model containing a filter.

CASE Filter Properties

For filter synthesis using SPLAT, two other properties are necessary. The `CASE_Properties::Filtering => 100` property association indicates that the component is a CASE Filter. The `CASE_Properties::Component_Spec` property association lists the AGREE specification IDs of the guarantee statements that comprise the filter expression. For example, in Figure

10, the `Component_Spec` property lists the identifier corresponding to the AGREE guarantee statement on line 54. The specification of `WELL_FORMED_LAT_LONG()` provides the definition of well-formedness for the filter.

Filter Synthesis

The filter implementation can be synthesized using the SPLAT tool, which will also provide a proof of correctness. The instructions for using SPLAT are described [below](#).

Design Assurance

It is crucial to have evidence of design correctness both at the time the model transformation is performed, and any time through system build. Resolute provides such assurance via augmentation of the requirement with assurance sub-claims as model transformations are performed.

When a formal requirement is imported from a TA1 tool, it will appear as in Figure 13.

```

4 goal Req_Wellformed_LatLong(comp_context : component, property_id : string) <=
5   ** "[well_formed_latlong] The Consumer shall only receive well-formed latitude-longitude coordinates" **
6   context Generated_By : "GearCASE";
7   context Generated_On : "May 23, 2020";
8   context Req_Component : "Producer_Consumer::SW.Impl.Consumer";
9   context Formalized : "True";
10  agree_prop_checked(comp_context, property_id)

```

Figure 13. Requirement imported from a TA1 tool.

Initially, there is not much for Resolute to check because the requirement hasn't yet been addressed in the design. All Resolute can do in this example is check that AGREE analysis was performed. Note that Resolute uses a separate plugin called AgreeCheck to determine if AGREE analysis was performed. AgreeCheck is included with Resolute but requires initial user configuration. In order to successfully use AgreeCheck, "Generate property analysis log" must be checked in the AGREE Analysis preferences, and a log file pathname must be specified. The AGREE Analysis preferences can be accessed by selecting Window → Preferences from the main menu, expanding the Agree node on the left-hand side of the preference window, and selecting Analysis.

Once the Filter transform is applied, the requirement is updated with an additional check to make, which reflects the addition of the filter component, as shown in Figure 14.

```

4 goal Req_Wellformed_LatLong(comp_context : component, property_id : string, filter : component, conn : connection, message_type : data) <=
5   ** "[well_formed_latlong] The Consumer shall only receive well-formed latitude-longitude coordinates" **
6   context Generated_By : "GearCASE";
7   context Generated_On : "May 23, 2020";
8   context Req_Component : "Producer_Consumer::SW.Impl.Consumer";
9   context Formalized : "True";
10  agree_prop_checked(comp_context, property_id) and add_filter(comp_context, filter, conn, message_type)

```

Figure 14. Modified requirement after Filter transform.

The addition of the `add_filter()` call on line 10 provides Resolute with additional checks to ensure the requirement was addressed correctly. `add_filter()` is included in the `CASE_Model_Transformations` library (which is included with BriefCASE) and consists of three subclaims:

- `filter_exists()` – Checks that the filter component is present in the model
- `filter_not_bypassed()` – Checks that there are no connections in the model that bypass the filter
- `component_implemented()` – Checks that the filter was implemented correctly

To check whether the requirement has been correctly addressed in the design, select the containing component implementation (SW.Impl) and select Analyses → Resolute from the main menu. The Resolute output will appear in the output pane, as shown in Figure 15.

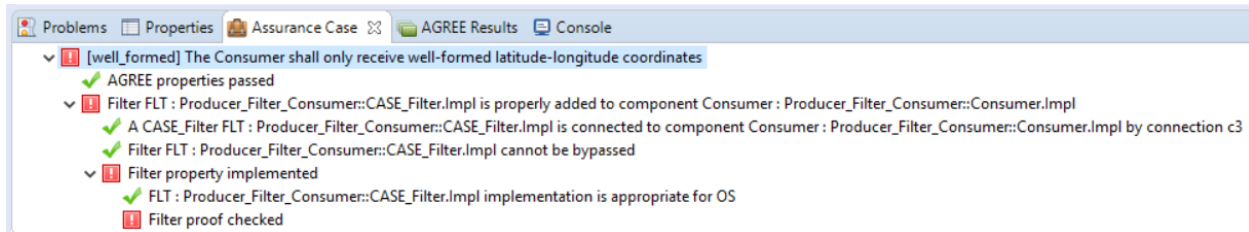


Figure 15. A failing Resolute analysis.

In this example, all checks passed except for the filter proof. Because SPLAT was not run on the current version of the model, the entire assurance case fails. By running SPLAT, all criteria are satisfied for addressing the requirement, and the Resolute output appears as in Figure 16.

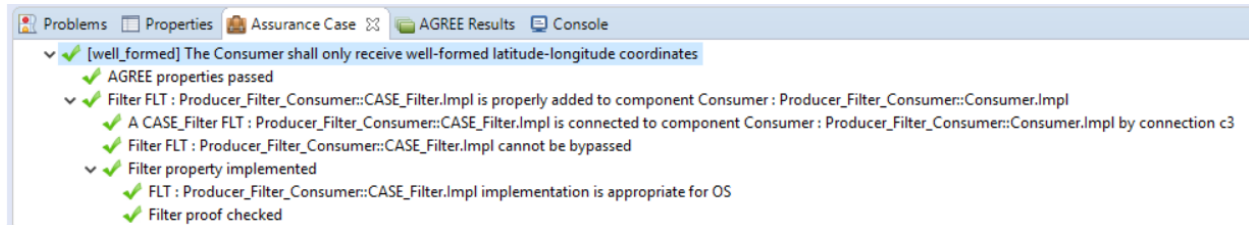


Figure 16. A passing Resolute analysis.

Attestation

To illustrate the Attestation transform, we use a simple producer-consumer example model. Both an initial model and a transformed model can be found here:

https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Attestation/Simple_Example

Two AADL packages are included:

- Attestation.aadl – This is the initial model.
- CASE_Requirements.aadl – This is the package containing the requirement (in the form of a Resolute claim) that drives the attestation transform.

CASE Attestation components are added to a communication driver component to ensure that only messages from trustworthy sources are accepted. Attestation consists of two components:

1. Attestation Manager – Performs remote attestation on a message source
2. Attestation Gate – Drops messages from sources that have not passed attestation

Only components with the **CASE_Properties::Comm_Driver => True** property association can have an Attestation Manager connected to it. All outgoing connections from the communication

driver will pass through the Attestation Gate. The Attestation Manager and Gate component types that are inserted into the model will be the same component type as the communication driver.

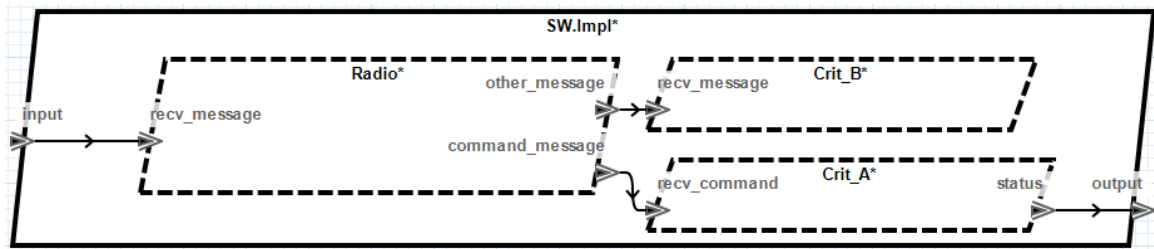
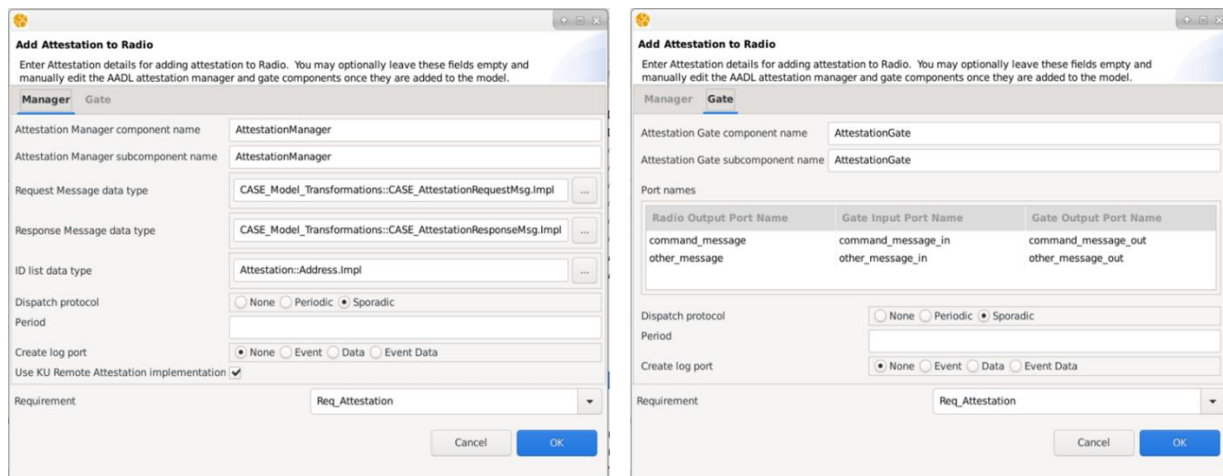


Figure 17. Initial model.

Attestation is added to the model by selecting a communication driver subcomponent in a component implementation (for example, in *Attestation_Simple_Initial/Attestation.aadl*, select the **Radio** subcomponent on line 68). Click the **BriefCASE** → **Cyber Resiliency** → **Model Transformations** → **Add Attestation...** menu item. A wizard will appear, as shown in Figure 18, enabling the user to customize the Attestation components. The wizard has two tabs; one for Attestation Manager configuration, and the other for Attestation Gate configuration.



The figure shows two screenshots of the 'Add Attestation to Radio' wizard. The left screenshot is the 'Attestation Manager' tab, and the right screenshot is the 'Attestation Gate' tab. Both tabs have a 'Manager' and a 'Gate' sub-tab. The 'Manager' sub-tab is selected in both. The 'Attestation Manager' tab has fields for 'Attestation Manager component name' (AttestationManager), 'Attestation Manager subcomponent name' (AttestationManager), 'Request Message data type' (CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl), 'Response Message data type' (CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl), 'ID list data type' (Attestation::Address.Impl), 'Dispatch protocol' (None, Periodic, Sporadic), 'Period' (empty), 'Create log port' (None, Event, Data, Event Data), 'Use KU Remote Attestation implementation' (checked), and 'Requirement' (Req_Attestation). The 'Attestation Gate' tab has fields for 'Attestation Gate component name' (AttestationGate), 'Attestation Gate subcomponent name' (AttestationGate), 'Port names' (Radio Output Port Name, Gate Input Port Name, Gate Output Port Name), 'Dispatch protocol' (None, Periodic, Sporadic), 'Period' (empty), 'Create log port' (None, Event, Data, Event Data), and 'Requirement' (Req_Attestation).

Figure 18. Add Attestation Manager wizard. Left: Attestation Manager tab. Right: Attestation Gate tab.

The Attestation transform will create Attestation Manager and Attestation Gate AADL component types and implementations, and insert them into the model. It will then instantiate the Attestation Manager and Gate as subcomponents in the implementation containing the selected communication driver. On both the Manager and Gate tabs, the user may provide a name for the attestation components and subcomponent instantiations, or use the default. If the field is left blank, the default name will be used. Note that if the specified name already exists, a number will be appended to the name to make it unique within the containing component implementation.

The Attestation Manager sends attestation request messages to, and receives attestation response messages from, the remote system undergoing attestation. The AADL message types can be entered into the wizard. The “...” button to the right of the text box provides a list of all data types visible to the current AADL package. The user may also enter a data type from other packages not yet imported.

Note that if the type is defined in a different package, the type name must be qualified with the package name in the `<package name>::<type name>` format.

The Attestation Manager will provide the Attestation Gate with a list of system IDs that have passed attestation and are therefore trusted. The user can specify the AADL data type of the trusted ID list, which the transform will then include on the connection connecting the two attestation components.

If the attestation components are threads, the user can specify the `Dispatch_Protocol` and `Period` properties for each of them.

By default the attestation components will drop any messages that do not originate from trusted sources, and no record of the malformed message will be retained. If the user wishes to log the event, an additional log port can be added to either component. The user will need to specify the AADL port type (Event, Data, or EventData) and it will be up to the user to connect the log port to an appropriate “logger” component and provide the implementation.

The Attestation Gate will have input and output ports corresponding to each incoming message from the communication driver. The user can set the names for these ports, or use the default names.

The requirement drop-down box lists all the imported cyber-requirements. By specifying the cyber requirement that drives the attestation transform, the appropriate assurance argument can be constructed for demonstrating that the requirement was addressed correctly. A requirement does not need to be selected to insert the attestation components, but it is highly recommended for construction of the proper system assurance case.

BriefCASE includes an Attestation Manager implementation developed by the University of Kansas (KU). By selecting to use the KU implementation, the attestation source code will be inserted into the project.

Clicking the OK button on the wizard will insert the Attestation Manager and Attestation Gate components into the model, as shown in Figure 19. Figure 20 shows the attestation subcomponent instantiations. Note that in addition to inserting the attestation components, the transform also made a copy of the communication driver implementation (see Figure 21). This is because two new connections (*AttestationRequest* and *AttestationResponse*) are required for the Attestation Manager. The graphical representation of the transformed model is shown in Figure 22.

```

74⊖  thread AttestationManager
75      features
76          AttestationRequest: out event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
77          AttestationResponse: in event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
78          TrustedIds: out event data port Address.Impl;
79      properties
80          CASE_Properties::Attesting => 100;
81  end AttestationManager;
82
83⊖  thread implementation AttestationManager.Impl
84      properties
85          Dispatch_Protocol => Sporadic;
86          CASE_Properties::Component_Language => CakeML;
87⊖  Source_Text => ("Component_Source/Attestation/build/heli_am.S",
88                  "Component_Source/Attestation/build/apps/case-tool-assessment/libheli_am_c.a");
89  end AttestationManager.Impl;
90
91⊖  thread AttestationGate
92      features
93          command_message_in: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
94          command_message_out: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
95          other_message_in: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
96          other_message_out: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
97          TrustedIds: in event data port Address.Impl;
98      properties
99          CASE_Properties::Gating => 100;
100  end AttestationGate;
101
102⊖  thread implementation AttestationGate.Impl
103      properties
104          Dispatch_Protocol => Sporadic;
105  end AttestationGate.Impl;

```

Figure 19. Line 74: Attestation Manager component type; Line 83: Attestation Manager component implementation; Line 91: Attestation Gate component type; Line 102: Attestation Gate component implementation.

```

116⊖ process implementation SW.Impl
117     subcomponents
118         Radio: thread RadioDriver_Attestation.Impl;
119         Crit_A: thread Critical_A.Impl;
120         Crit_B: thread Critical_B.Impl;
121         AttestationManager: thread AttestationManager.Impl;
122         AttestationGate: thread AttestationGate.Impl;
123     connections
124         c1: port input -> Radio.recv_message;
125         c5: port Radio.command_message -> AttestationGate.command_message_in;
126         c6: port Radio.other_message -> AttestationGate.other_message_in;
127         c7: port AttestationManager.TrustedIds -> AttestationGate.TrustedIds;
128         c8: port AttestationManager.AttestationRequest -> Radio.AttestationRequest;
129         c9: port Radio.AttestationResponse -> AttestationManager.AttestationResponse;
130         c2: port AttestationGate.command_message_out -> Crit_A.recv_command;
131         c3: port AttestationGate.other_message_out -> Crit_B.recv_message;
132         c4: port Crit_A.status -> output;
133⊖  annex resolute {**
134      prove Req_Attestation(this.Radio, this.AttestationManager, this.AttestationGate)
135      **};
136  end SW.Impl;

```

Figure 20. Line 118: New communication driver component; Line 121: Attestation Manager subcomponent; Line 122: Attestation Gate subcomponent; Lines 125-131: Attestation connections; Line 134: updated assurance claim call.

```

77 thread RadioDriver_Attestation
78   features
79     recv_message: in event data port;
80     command_message: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
81     other_message: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
82     AttestationRequest: in event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
83     AttestationResponse: out event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
84   properties
85     CASE_Properties::Comm_Driver => true;
86 end RadioDriver_Attestation;

```

Figure 21. Line 57: New comm driver definition with attestation ports (lines 62,63).

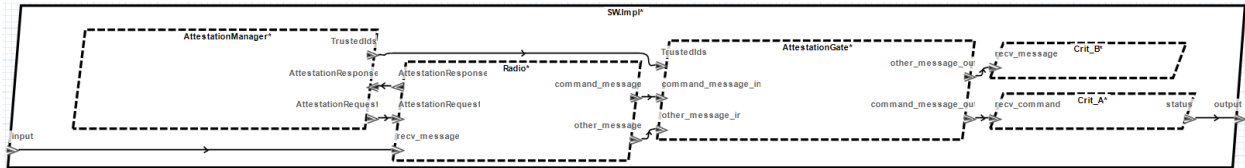


Figure 22. Transformed model containing an attestation manager.

For every outgoing connection from the communication driver to other software components, a corresponding input and output port is created in the attestation gate. The Attestation transform also adds two connections between the Attestation Manager and the communication driver to perform the attestation with the message source. Because the communication driver implementation may be instantiated in other parts of the system, a new communication driver with the additional attestation ports is created that extends the original communication driver.

Design Assurance

It is crucial to have evidence of design correctness both at the time the model transformation is performed, and at any time up through system build. Resolute provides that assurance via augmentation of the requirement with assurance sub-claims as model transformations are performed.

When a cyber requirement is imported it will be placed in the CASE_Requirements package as a Resolute claim, as shown in Figure 23.

```
4 goal Req_Attestation(comp_context : component) <=
5   ** "[attestation] Only messages from trusted sources shall be accepted" **
6   context Generated_By : "GearCASE";
7   context Generated_On : "Jul 20, 2020";
8   context Req_Component : "Attestation::SW.Impl.Radio";
9   context Formalized : "False";
10  undeveloped
```

Figure 23. Requirement imported from a TA1 tool.

Initially, there is nothing for Resolute to check because the requirement hasn't yet been addressed in the design. Once the Attestation transform is applied, the requirement is updated with a new check to be made, which reflects the addition of the attestation manager component, as shown in Figure 24.

```

5 goal Req_Attestation(comp_context : component, attestation_manager : component, attestation_gate : component) <=
6   ** "[attestation] Only messages from trusted sources shall be accepted" **
7   context Generated_By : "GearCASE";
8   context Generated_On : "Jul 20, 2020";
9   context Req_Component : "Attestation::SW.Impl.Radio";
10  context Formalized : "False";
11  add_attestation_manager(comp_context, attestation_manager, attestation_gate)

```

Figure 24. Modified requirement after Attestation transform.

The addition of the `add_attestation_manager()` call on line 11 provides Resolute with additional checks to make to ensure the requirement was addressed correctly.

`add_attestation_manager()` is included in the `CASE_Model_Transformations` library and consists of three sub-claims:

- `attestation_manager_exists()` – Checks that the attestation manager component is present in the model
- `attestation_manager_not_bypassed()` – Checks that there are no connections in the model that bypass the attestation manager
- `component_implemented()` – Checks that the attestation components were implemented correctly

To check whether the requirement has been correctly addressed in the design, select the containing component implementation (SW.Impl on line 116 in `Attestation.aadl`) and select **Analyses** → **Resolute** from the main menu. The Resolute output will appear in the output pane, as shown in Figure 25.

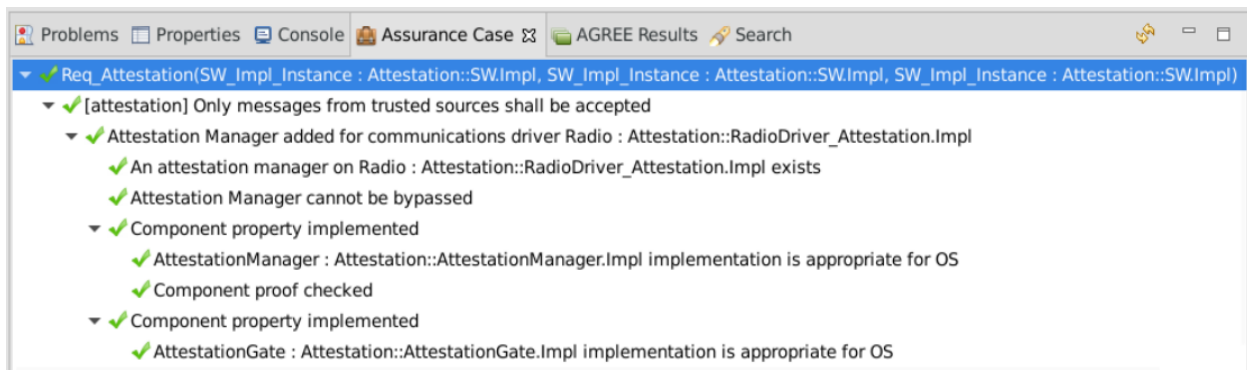


Figure 25. A passing Resolute analysis.

Virtualization

Note: The Virtualization transform was previously referred to as the Isolator transform.

To illustrate the Virtualization transform, we use a simple single-process example model, which can be found here:

[https://github.com/loonwerks/CASE/tree/master/TA2/Model Transformations/Virtualization/Simple Example](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/Virtualization/Simple%20Example)

Two AADL packages are included:

- `Virtualization.aadl` – This is the initial model.

- CASE_Requirements.aadl – This is the package containing the requirement (in the form of a Resolute claim) that drives the Virtualization transform.

Software systems and processes can be bound to a virtual machine. Note that to virtualize software components using the Virtualization model transformation, they must already be bound to a processor component. For example, the SW component is bound to the PROC component on line 75 of Virtualization.aadl (see Figure 26).

```

70 system implementation Critical.Impl
71   subcomponents
72     PROC : processor HW_Proc.Impl;
73     SW : process SW.Impl;
74   properties
75     Actual_Processor_Binding => (reference (PROC)) applies to SW.
76 annex resolute {**
77   prove(Req_Virtualization(this.SW))
78 **};
79 end Critical.Impl;

```

Figure 26. Line 75: Software process is bound to hardware processor component via Actual_Processor_Binding property association.

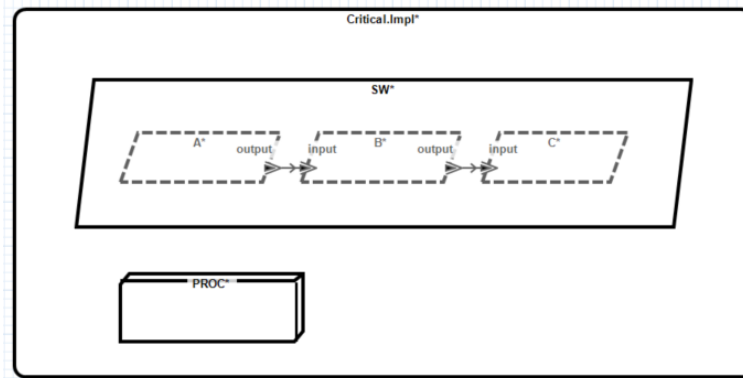


Figure 27. Initial model.

To apply the Virtualization transform, select a software system or process subcomponent in a system component implementation (for example, in Virtualization_Simple_Initial/Virtualization.aadl, select the SW subcomponent on line 73). Note that currently the transformation can only be applied from within the OSATE text editor (future versions will enable applying the transformation from within the graphical editor). Click the BriefCASE → Cyber Resiliency → Model Transformations → Add Virtualization... menu. A wizard will appear, as shown in Figure 28.

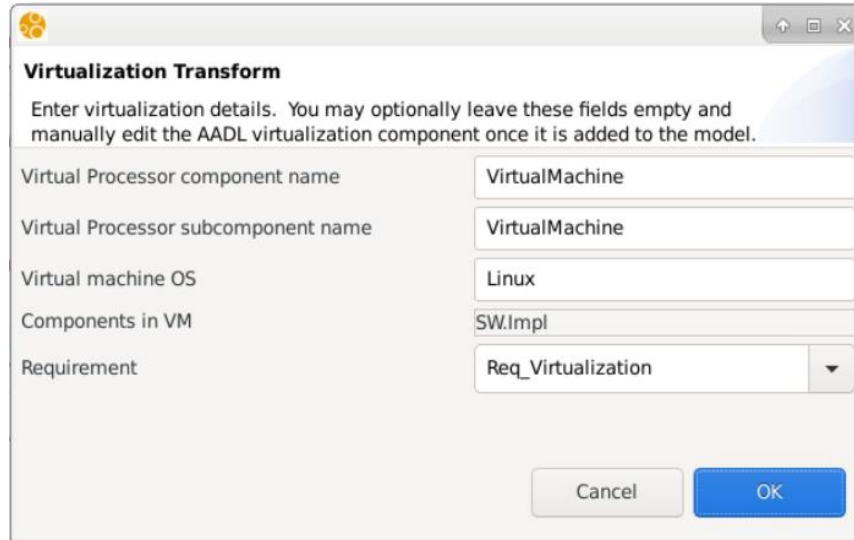


Figure 28. Add Virtualization wizard.

The Virtualization transform will create a new AADL virtual processor component and bind it to the same processor that the selected subcomponent was bound to. You can provide the name of the virtual processor component and instantiated subcomponent, or use the default names in the first two fields of the wizard. If either field is left blank, the default name will be used ('VirtualMachine' for the component type name and 'VirtualMachine' for the subcomponent name). Note that if the specified name already exists, a number will be appended to the name to make it unique within the containing component implementation.

The System Build process will package virtualized components in a virtual machine. You can specify the virtual machine operating system, use the default name, or leave it blank. If the selected component is a software system containing multiple processes, the Virtualization transform enables you to choose whether you would like to bind the selected component and all of its subcomponents, or only specific subcomponents. Choosing to bind only selected subcomponents will enable checkboxes for each subcomponent for selection/de-selection.

The requirement drop-down box lists all the imported cyber-requirements. By specifying the cyber requirement that drives the Virtualization transform, the appropriate assurance argument can be constructed for demonstrating the requirement was addressed correctly. A requirement does not need to be selected to add virtualization, but it is highly recommended for construction of the proper system assurance case.

Clicking OK will close the wizard and apply the model transformation. A VirtualMachine component type and component implementation are added to the AADL file, and a VirtualMachine subcomponent is inserted into the component implementation containing the software component that was selected for virtualization (see Figure 29). The graphical representation is shown in Figure 30.

```

70 virtual processor VirtualMachine
71   properties
72     CASE_Properties::Isolating => 100;
73 end VirtualMachine;
74
75 virtual processor implementation VirtualMachine.Impl
76   properties
77     CASE_Properties::OS => Linux;
78 end VirtualMachine.Impl;
79
80 system Critical
81 end Critical;
82
83 system implementation Critical.Impl
84   subcomponents
85     PROC: processor HW_Proc.Impl;
86     SW: process SW.Impl;
87     VirtualMachine: virtual processor VirtualMachine.Impl;
88   properties
89     Actual_Processor_Binding => (reference (PROC)) applies to VirtualMachine;
90     Actual_Processor_Binding => (reference (VirtualMachine)) applies to SW;
91 annex resolute {**
92   prove Req_Virtualization(this.SW, {this.SW}, this.VirtualMachine)
93 **};
94 end Critical.Impl;

```

Figure 29. Line 70: VirtualMachine component type; Line 75: VirtualMachine component implementation; Line 87: VirtualMachine subcomponent; Lines 89-90: updated processor bindings.

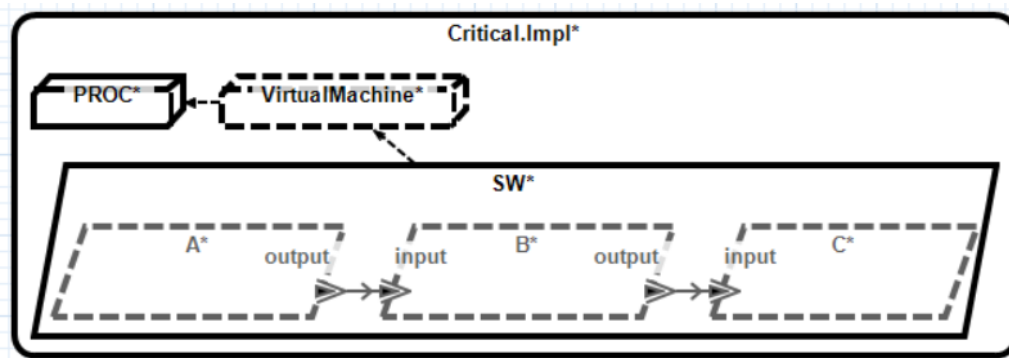


Figure 30. Transformed model containing a virtual machine.

Virtualization is represented in AADL by binding a virtual processor component to a processor component, and then binding selected software components to the virtual processor. The transform will also remove existing bindings between the selected software components and the processor component they were originally bound to. Note that per AADL semantics, if a component implementation is bound to a processor, that binding is also inherited by that component's subcomponents, unless a subcomponent has an explicit binding to a different processor.

Design Assurance

As part of the transform, the requirement (specified in the model as a Resolute claim) will be updated with an `add_virtualization` sub-claim from the `CASE_Model_Transformations` claim library (see Figure 31). This will provide assurance that the model transformation was performed correctly, and that the processor bindings are preserved throughout the remainder of system design, and through every step of the build process.

```

5 goal Req_Virtualization(comp_context : component, vm_components : {component}, virtual_machine : component) <=
6   ** "[virtualization] Third-party software shall be isolated from critical components" **
7   context Generated_By : "GearCASE";
8   context Generated_On : "May 23, 2020";
9   context Req_Component : "Virtualization::Critical.Impl.SW";
10  context Formalized : "False";
11  add_virtualization(vm_components, virtual_machine)

```

Figure 31. Virtualization requirement in Resolute.

The addition of the `add_virtualization()` call on line 10 provides Resolute with additional checks to ensure the requirement was addressed correctly. `add_virtualization()` is included in the `CASE_Model_Transformations` library and consists of three sub-claims:

- `vm_bound_to_processor()` – Checks that the virtual processor is bound to a processor
- `components_bound_to_vm()` – Checks that the selected components are bound to the virtual processor
- `subcomponents_not_bound_to_other_processors()` – Checks that there are no subcomponents of selected components that are bound to a difference processor

To check whether the requirement has been correctly addressed in the design, select the containing component implementation (Critical.Impl on line 83 in Virtualization.aadl) and select Analyses → Resolute from the main menu. The Resolute output will appear in the output pane, as shown in Figure 32.

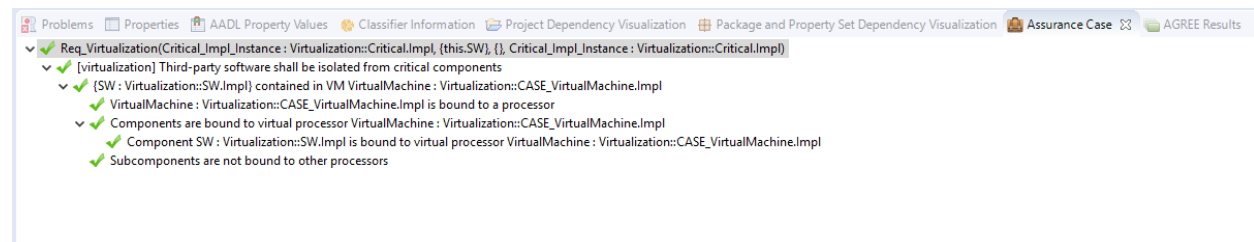


Figure 32. A passing Resolute analysis.

Monitor

To illustrate the Monitor transform, we use a simple producer-consumer example model, which can be found here:

https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Monitor/Simple_Example

Two AADL packages are included:

- Monitor.aadl – This is the initial model.

- CASE_Requirements.aadl – This is the package containing the requirement (in the form of a Resolute claim) that drives the monitor transform.

A CASE Monitor is added to an AADL connection to provide an alert when a policy on the connection's data has been violated (or alternatively, satisfied). In addition to signaling an alert when the monitor policy has been violated, the monitor can also be used as a gate to prevent data from propagating. Multiple monitors may be placed on the same connection. The Monitor component type that is inserted into the model will be the same component type as the connection source, with two exceptions:

1. If the destination component is a thread group, the monitor will be a thread.
2. If the destination component is a process containing a single thread, the monitor will also be a process containing a single thread.

The latter supports the seL4 representation of components, in which each thread runs in its own address space. Note that for System Build, the monitor must be a software component (either a thread or process containing a single thread).

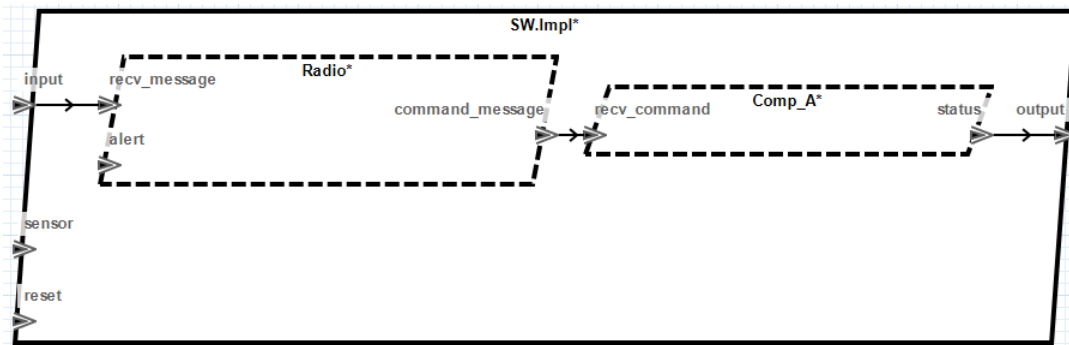


Figure 33. Initial model.

A monitor can be added to the following AADL components:

- Thread
- Thread Group
- Process containing a single thread

This example describes how to insert a response monitor on a component (Comp_A in Monitor.aad). The response monitor observes the output of Comp_A, expecting a response within two timesteps after receiving a command. A second example demonstrates a gated monitor.

To insert the response monitor, select the connection in a component implementation that requires monitoring (for example, in Monitor_Simple_Initial/Monitor.aadl, select the c3 connection on line 54). Note that currently the transform can only be applied from within the OSATE text editor (future versions will enable applying the transformation from within the graphical editor). In the main menu, click the BriefCASE → Cyber Resiliency → Model Transformations → Add Monitor... menu item. A wizard will

open, as shown in Figure 34. The wizard enables the user to customize the monitor, including providing the monitor policy specification.

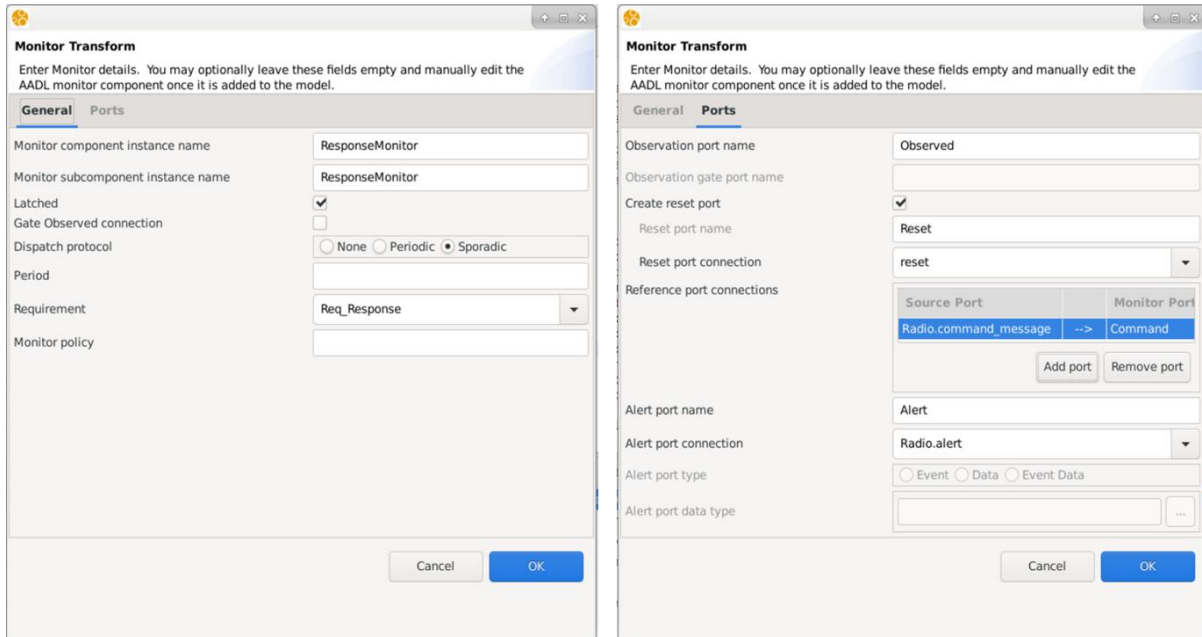


Figure 34. Add Monitor wizard for Response Monitor.

The Monitor transform will create a Monitor AADL component type and implementation, and insert them into the model. It will then instantiate the Monitor implementation as a subcomponent in the implementation containing the selected connection. The user may provide the names for the monitor component and subcomponent, or use the default ('Monitor' for the component type and the subcomponent). If the field is left blank, the default name will be used. Note that if the specified name already exists, a number will be appended to the name to make it unique within the containing component implementation.

Runtime monitors can be *latched*, meaning once the monitor policy has been violated the alert signal will remain high until the monitor is *reset*. A reset port can be added to the component by checking the box. This will enable the dropdown box containing other ports in the model with which the monitor reset port can be connected. If a source reset port is selected, the monitor reset port will be of the same type (Data, Event, Event Data). It is not necessary to select a specific reset source port to add a reset port.

Monitors typically function by comparing an observed signal with one or more reference signals. The *Reference port connections* input provides the ability to create monitor reference ports, select the source component port to connect to, and provide a name for the reference port.

The *Alert port connection* dropdown box contains other ports in the model to connect the monitor alert port to. The monitor alert port will be the same type as the destination alert port. It is not necessary to

select a specific alert destination port, in which case the monitor alert port will be an event data port with no data type. If an alert port connection is not specified, the user may specify the data type.

By selecting the *Gate Observed connection* checkbox, a monitor will be generated that not only observes a specific connection, but routes the connection through the monitor. In this configuration, when the monitor policy is violated, the monitor can prevent the propagation of the observed signal. An example of creating a gated monitor is provided below.

The requirement drop-down box lists all the cyber-requirements that have been imported into the model. By specifying the cyber requirement that drives the Monitor transform, the appropriate assurance argument can be constructed for demonstrating the requirement was addressed correctly. A requirement does not need to be selected to insert the monitor, but it is highly recommended for construction of the proper system assurance case.

Finally, the user may provide the *Monitor policy*, in the form of an AGREE statement. The policy should evaluate to a Boolean value, and typically compares the observed signal to constant values or reference signals. Note that no syntax validation is performed on the AGREE statement. If it is malformed, it may not be imported into the model properly. The user may also leave this field blank and enter the policy directly in the model after the transform is performed.

Clicking the OK button on the wizard will insert the Monitor into the model, as shown in Figure 35 and Figure 36. The graphical representation is shown in Figure 37.

```

36-  thread ResponseMonitor
37      features
38          Observed: in event data port Base_Types::Boolean;
39          Command: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
40          Alert: out event data port Base_Types::Boolean;
41          Reset: in event data port Base_Types::Boolean;
42      properties
43          CASE_Properties::Monitoring => 100;
44  end ResponseMonitor;
45
46-  thread implementation ResponseMonitor.Impl
47      properties
48          Dispatch_Protocol => Sporadic;
49  end ResponseMonitor.Impl;

```

Figure 35. Line 36: Response monitor component type; Line 46: Response monitor component implementation.

```

62 process implementation SW.Impl
63   subcomponents
64     Radio: thread RadioDriver.Impl;
65     Comp_A: thread Component_A.Impl;
66     ResponseMonitor: thread ResponseMonitor.Impl;
67   connections
68     c1: port input -> Radio.recv_message;
69     c2: port Radio.command_message -> Comp_A.recv_command;
70     c3: port Comp_A.status -> output;
71     c4: port Comp_A.status -> ResponseMonitor.Observed;
72     c5: port Radio.command_message -> ResponseMonitor.Command;
73     c6: port ResponseMonitor.Alert -> Radio.alert;
74     c7: port reset -> ResponseMonitor.Reset;
75 annex resolute {**
76   prove Req_CorrectOutput(this.Comp_A)
77   prove Req_Response(this.Comp_A, this.ResponseMonitor, this.ResponseMonitor.Alert)
78   **};
79 end SW.Impl;

```

Figure 36. Line 66: Response monitor subcomponent; Lines 71-74: Response monitor connections; Line 77: updated assurance claim call.

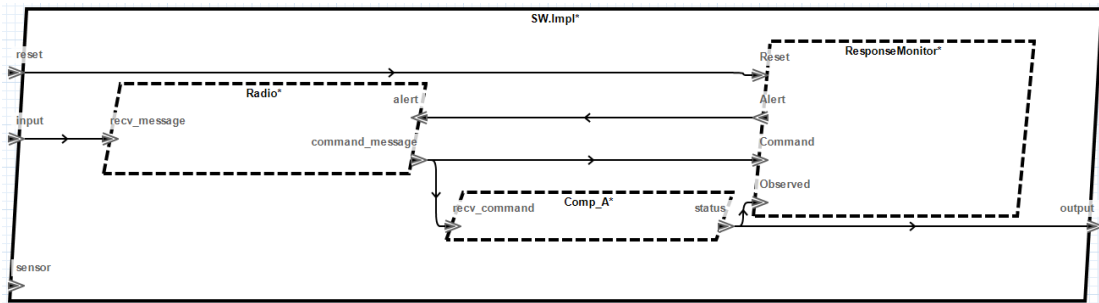


Figure 37. Transformed model containing a response monitor.

Although the monitor policy was not provided in the wizard, it can still be added to the model. For a response monitor, the following AGREE annex can be added to the ResponseMonitor component type:

```

annex agree {**

  -- Monitor policy (models the expected behavior in terms of the input
  ports)

  -- status (Observed) occurs within two time steps of receiving a
  command (Command)

  const is_latched : bool = false;

  eq empty_day : bool = not event(Observed) and not event(Command);

  property ResponseMonitor_policy = Historically(event(Command) or
  (empty_day and Yesterday(event(Command) or (empty_day and
  Yesterday(event(Command)))))) => event(Observed));

  property alerted = (not ResponseMonitor_policy) -> ((is_latched and
  pre(alerted)) or (event(Observed) and not ResponseMonitor_policy));

```

```

    guarantee ResponseMonitor_Alert "A violation of the monitor policy
shall trigger an alert" : event(Alert) <=> alerted;

**};

```

In addition, the component specification identifier will need to be referenced in the AADL property as:

```
CASE_Properties::Component_Spec => ("ResponseMonitor_Alert");
```

The specification is used by SPLAT to synthesize the monitor component. The instructions for using SPLAT are described [below](#). After inserting the above monitor policy, the response monitor will look similar to Figure 38.

```

36  thread ResponseMonitor
37      features
38          Observed: in event data port Base_Types::Boolean;
39          Command: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
40          Alert: out event data port Base_Types::Boolean;
41          Reset: in event data port Base_Types::Boolean;
42      properties
43          CASE_Properties::Monitoring => 100;
44          CASE_Properties::Component_Spec => ("ResponseMonitor_Alert");
45      annex agree {**
46          -- Monitor policy (models the expected behavior in terms of the input ports)
47          -- status (Observed) occurs within two time steps of receiving a command (Command)
48          const is_latched : bool = false;
49          eq empty_day : bool = not event(Observed) and not event(Command);
50          property ResponseMonitor_policy = Historically(event(Command) or
51              (empty_day and
52                  Yesterday(event(Command) or (empty_day and Yesterday(event(Command))))
53              ) => event(Observed)
54          );
55          property alerted = (not ResponseMonitor_policy) ->
56              ((is_latched and pre(alerted)) or (event(Observed) and not ResponseMonitor_policy));
57          guarantee ResponseMonitor_Alert "A violation of the monitor policy shall trigger an alert" :
58              event(Alert) <=> alerted;
59      **};
60  end ResponseMonitor;

```

Figure 38. Lines 45-59: Response monitor policy.

To insert a *gated* monitor, the process is similar to the one above. This monitor will observe the output of Comp_A (connection c3 of SW.Impl) and prevent it from propagating if it doesn't match an external sensor signal. For this monitor, the 'Gate Observed Connection' checkbox is checked, as shown in Figure 39.

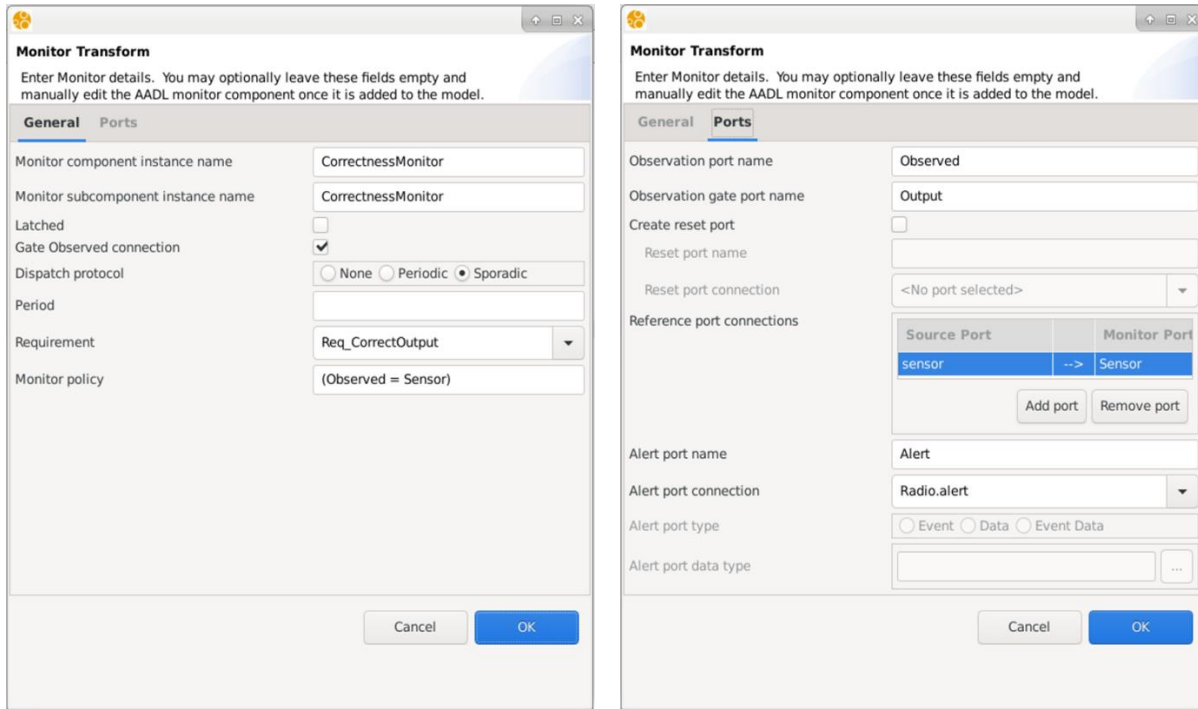


Figure 39. Wizard for adding a gated monitor.

Clicking the OK button on the wizard will insert the Monitor into the model, as shown in Figure 40 and Figure 41. The graphical representation is shown in Figure 42.

```

67@  thread CorrectnessMonitor
68      features
69          Observed: in event data port Base_Types::Boolean;
70          Sensor: in event data port Base_Types::Boolean;
71          Alert: out event data port Base_Types::Boolean;
72          Output: out event data port Base_Types::Boolean;
73      properties
74          CASE_Properties::Monitoring => 100;
75          CASE_Properties::Component_Spec => ("CorrectnessMonitor_Alert", "CorrectnessMonitor_Output");
76@  annex agree {**
77@      const is_latched : bool = false;
78      property alerted = (not (Observed = Sensor)) -> ((is_latched and pre(alerted)) or (event(Observed) and not (Observed = Sensor)));
79      guarantee CorrectnessMonitor_Alert "A violation of the monitor policy shall trigger an alert" : event(Alert) <=> alerted;
80@      guarantee CorrectnessMonitor_Output "A violation of the monitor policy shall prevent propagation of the observed input." :
81          if alerted then not event(Output) else event(Output) and Output = Observed;
82      **};
83  end CorrectnessMonitor;
84
85@  thread implementation CorrectnessMonitor.Impl
86      properties
87          Dispatch_Protocol => Sporadic;
88  end CorrectnessMonitor.Impl;
--

```

Figure 40. Line 67: Correctness monitor component type; Line 85: Correctness monitor component implementation.


```

101 process implementation SW.Impl
102   subcomponents
103     Radio: thread RadioDriver.Impl;
104     Comp_A: thread Component A.Impl;
105     ResponseMonitor: thread ResponseMonitor.Impl;
106     CorrectnessMonitor: thread CorrectnessMonitor.Impl;
107   connections
108     c1: port input -> Radio.recv_message;
109     c2: port Radio.command_message -> Comp_A.recv_command;
110     c3: port CorrectnessMonitor.Output -> output;
111     c4: port Comp_A.status -> ResponseMonitor.Observed;
112     c5: port Radio.command_message -> ResponseMonitor.Command;
113     c6: port ResponseMonitor.Alert -> Radio.alert;
114     c7: port reset -> ResponseMonitor.Reset;
115     c8: port Comp_A.status -> CorrectnessMonitor.Observed;
116     c9: port sensor -> CorrectnessMonitor.Sensor;
117     c10: port CorrectnessMonitor.Alert -> Radio.alert;
118 annex resolute {**
119   prove Req_Response(this.Comp_A, this.ResponseMonitor, this.ResponseMonitor.Alert)
120   prove Req_CorrectOutput(this.Comp_A, this.CorrectnessMonitor, this.CorrectnessMonitor.Alert, this, Base_Types::Boolean)
121   **};
122 end SW.Impl;

```

Figure 41. Line 106: Correctness monitor subcomponent; Lines 115-117: Correctness monitor connections; Line 120: updated assurance claim call.

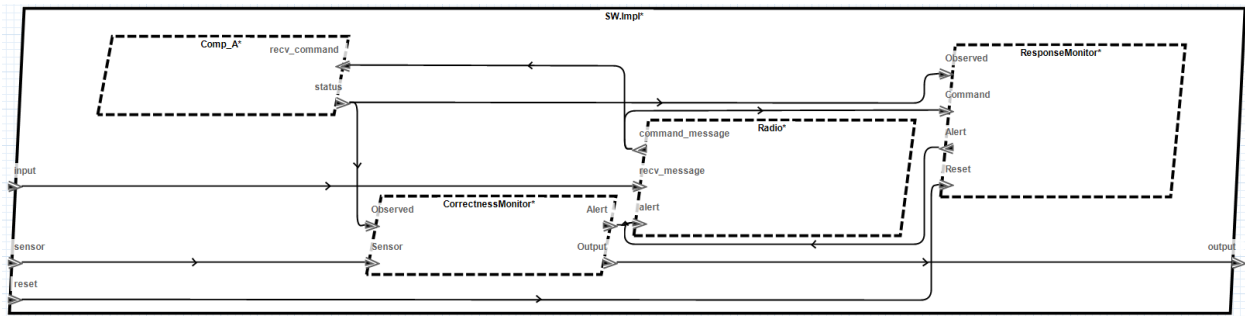


Figure 42. Transformed model containing a correctness monitor.

Design Assurance

It is crucial to have evidence of design correctness both at the time the model transformation is performed, and at any time up through system build. Resolute provides that assurance via augmentation of the requirement with assurance sub-claims as model transformations are performed.

When requirements are imported from a cyber requirements tool, they will be placed in the CASE_Requirements package as Resolute claims, as shown in Figure 43.


```

4 goal Req_Response(comp_context : component) <=
5   ** "[response_monitor] Component Comp_A shall be monitored for responsiveness" **
6   context Generated_By : "GearCASE";
7   context Generated_On : "Jan 29, 2020";
8   context Req_Component : "Monitor::SW.Impl.Comp_A";
9   context Formalized : "False";
10  undeveloped
11
12 goal Req_CorrectOutput(comp_context : component) <=
13   ** "[correctness_monitor] Component Comp_A shall be monitored for correct output" **
14   context Generated_By : "GearCASE";
15   context Generated_On : "Jan 29, 2020";
16   context Req_Component : "Monitor::SW.Impl.Comp_A";
17   context Formalized : "False";
18  undeveloped

```

Figure 43. Imported requirements.

Once the Monitor transforms have been applied, the requirements are updated with an additional check, which reflects the addition of the monitor components, as shown in Figure 44.

```

5 goal Req_Response(comp_context : component, monitor : component, alert_port : feature) <=
6   ** "[response_monitor] Component Comp_A shall be monitored for responsiveness" **
7   context Generated_By : "GearCASE";
8   context Generated_On : "Jan 29, 2020";
9   context Req_Component : "Monitor::SW.Impl.Comp_A";
10  context Formalized : "False";
11  add_monitor(comp_context, monitor, alert_port)
12
13 goal Req_CorrectOutput(comp_context : component, monitor : component, alert_port : feature, gate_context : component, message_type : data) <=
14   ** "[correctness_monitor] Component Comp_A shall be monitored for correct output" **
15   context Generated_By : "GearCASE";
16   context Generated_On : "Jan 29, 2020";
17   context Req_Component : "Monitor::SW.Impl.Comp_A";
18   context Formalized : "False";
19  add_monitor_gate(comp_context, monitor, alert_port, gate_context, message_type)

```

Figure 44. Modified requirements after Monitor transform.

The addition of the `add_monitor()` call on line 11 provides Resolute with an additional check to be made to ensure the requirement was addressed correctly. `add_monitor()` is included in the `CASE_Model_Transformations` library and consists of four sub-claims:

- `component_exists()` – Checks that the monitor component is present in the model
- `alert_connected()` – Checks that the monitor alert port is connected in the model
- `independent_reset()` – Checks that the monitor reset port cannot be set by the source of the observed signal
- `component_implemented()` – Checks that the monitor was correctly implemented for the appropriate OS

For gated monitors, a slightly different sub-claim, `add_monitor_gate()`, is inserted (line 19). This is because when a connection is gated, there is an additional assurance check that must be made.

`add_monitor_gate()` is also included in the `CASE_Model_Transformations` library and consists of one additional sub-claim:

- `component_not_bypassed()` – Checks that the observed signal being gated by the monitor cannot reach its destination via any other pathway in the system

To check whether the requirements have been correctly addressed in the design, select the containing component implementation (SW.Impl on line 101 in Monitor.aadl) and select Analyses → Resolute from the main menu. The Resolute output will appear in the output pane, as shown in Figure 45.

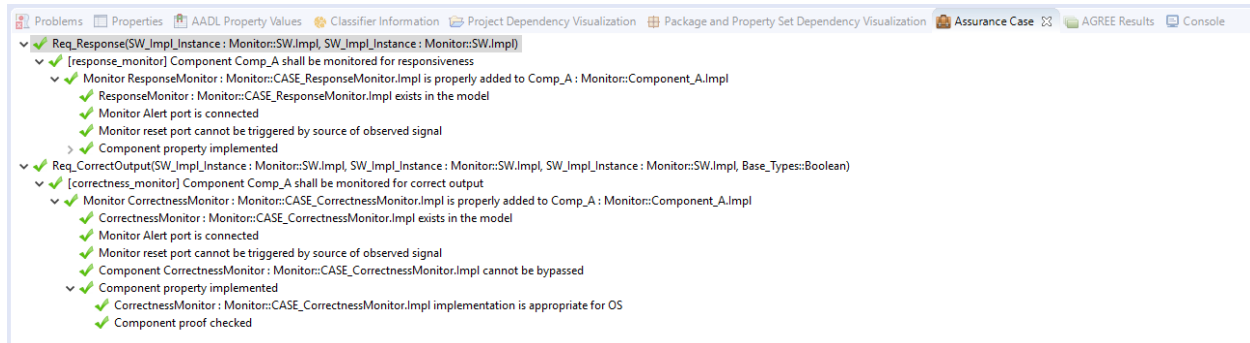


Figure 45. A passing Resolute analysis.

seL4

To illustrate the seL4 transform, we use a simple model containing several software component types. Both an initial model and a transformed model can be found here:

[https://github.com/loonwerks/CASE/tree/master/TA2/Model Transformations/seL4/Simple_Example](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/seL4/Simple_Example)

Two AADL packages are included:

- MissionComputer.aadl – This is the initial model of a mission computer containing both hardware and software.
- Software.aadl – This is the initial model of the software.
- CASE_Requirements.aadl – This is the package containing the requirement (in the form of a Resolute claim) that drives the filter transform.

The seL4 kernel runtime enforces component isolation such that each component is essentially a single thread running in its own process. In order to accurately reflect the seL4 runtime in an AADL model, the seL4 transform can be applied to models containing processes with thread groups or multiple threads, resulting in a model that is representative of the eventual HAMR seL4 build. For example, the model in Figure 46 depicts a software system, *SW_Sys*, containing two processes, *NonCritical* and *Critical*. The *Critical* process contains a thread, *Component_A*, and a thread group, *Component_E*, which itself contains three threads, *Component_B*, *Component_C*, and *Component_D*. However, if this system were implemented on seL4, each of the threads would run in their own address spaces (processes), so this model is inaccurate, which could lead to incorrect analysis during design.

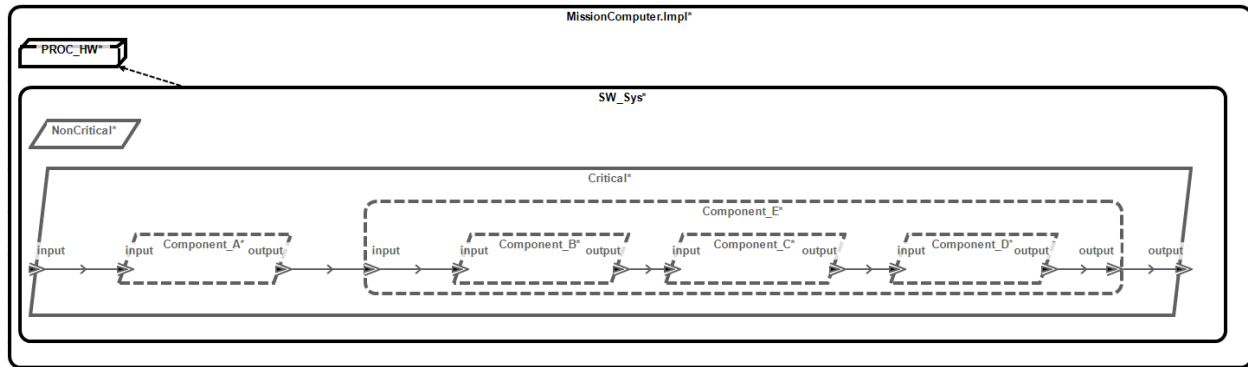


Figure 46. Initial model.

The seL4 transform can be performed on the following AADL components:

- Process
- Software system (a system component containing only system, process, thread group, and thread subcomponents)

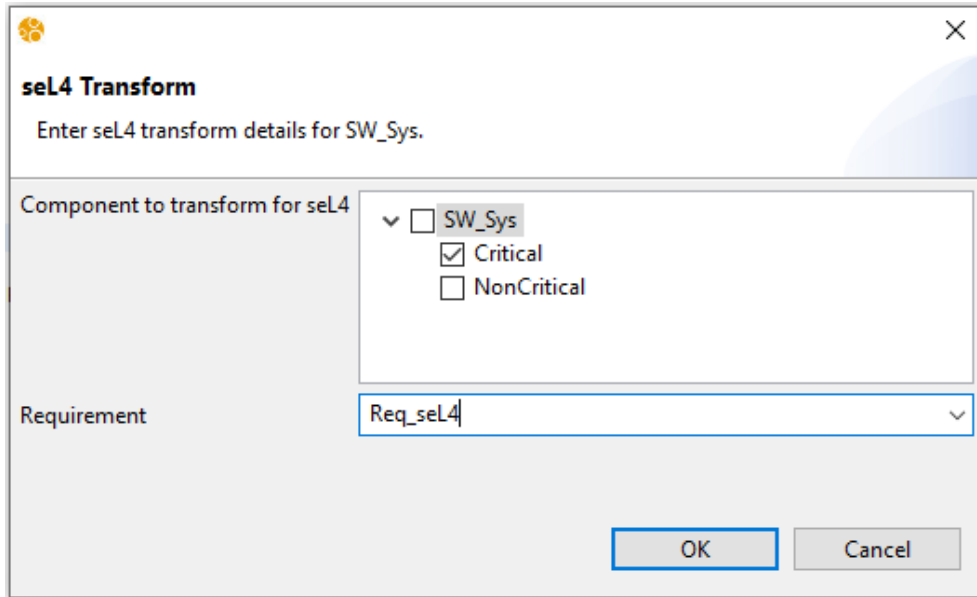
The seL4 transform will result in a subsystem in which:

- A process component is created for each thread, and the thread becomes the only subcomponent of that process
- A system component is created for each thread group, containing a process for each transformed thread
- A system component is created for each process, and the processes become subcomponents of that system
- Proper wiring of transformed components is managed
- Property associations are maintained (when applicable) on transformed components
- AGREE statements are copied or lifted

Note that AADL feature groups are not currently supported but will be in future versions of the tool.

To apply the seL4 transform, select a software system or process subcomponent in a system component implementation (for example, in seL4_Simple_Initial/MissionComputer.aadl, select the SW_Sys subcomponent on line 22). Note that currently the transformation can only be applied from within the OSATE text editor (future versions will enable applying the transformation from within the graphical

editor). Click the BriefCASE → Cyber Resiliency → Model Transformations → Transform for seL4 Build... menu item. A wizard will appear, as shown in Figure 47.



The image shows a dialog box titled "seL4 Transform" with a close button (X) in the top right corner. Below the title bar, it says "Enter seL4 transform details for SW_Sys." The dialog is divided into two main sections. The top section is labeled "Component to transform for seL4" and contains a tree view with a dropdown arrow. The tree view shows "SW_Sys" with a checkbox, which is expanded to show two sub-items: "Critical" (checked) and "NonCritical" (unchecked). The bottom section is labeled "Requirement" and contains a text box with the value "Req_sel4" and a dropdown arrow. At the bottom right of the dialog are two buttons: "OK" and "Cancel".

Figure 47. seL4 Transform wizard.

If a software system subcomponent is selected that contains multiple components, the wizard will display a selection list for choosing the component to transform. Note that only one system or process component can be transformed at a time.

The requirement drop-down box lists all the cyber-requirements that have been imported. By specifying the cyber requirement that drives the seL4 transformation, the appropriate assurance argument can be constructed for demonstrating the requirement was addressed correctly. A requirement does not need to be selected to perform the transform, but it is highly recommended for construction of the proper system assurance case.

Clicking the OK button on the wizard will transform the selected component (along with its descendants) in the model, as shown in Figure 48. The graphical representation is shown in Figure 49.

```

176- process Component_D_sel4
177-     features
178-         input: in event data port Base_Types::Integer;
179-         output: out event data port Base_Types::Integer;
180-     end Component_D_sel4;
181
182- process implementation Component_D_sel4.Impl
183-     subcomponents
184-         Component_D: thread Component_D.Impl;
185-     connections
186-         c1: port input -> Component_D.input;
187-         c2: port Component_D.output -> output;
188-     annex agree {**
189-         lift contract;
190-     **};
191- end Component_D_sel4.Impl;
192
193- system Component_E_sel4
194-     features
195-         input: in event data port Base_Types::Integer;
196-         output: out event data port Base_Types::Integer;
197-     annex agree {**
198-         assume Req001_E_sel4 "Input is positive" : POSITIVE(input);
199-         guarantee Req002_E_sel4 "Output is positive" : POSITIVE(output);
200-     **};
201- end Component_E_sel4;
202
203- system implementation Component_E_sel4.Impl
204-     subcomponents
205-         Component_D: process Component_D_sel4.Impl;
206-         Component_C: process Component_C_sel4.Impl;
207-         Component_B: process Component_B_sel4.Impl;
208-     connections
209-         c1: port input -> Component_B.input;
210-         c2: port Component_B.output -> Component_C.input;
211-         c3: port Component_C.output -> Component_D.input;
212-         c4: port Component_D.output -> output;
213- end Component_E_sel4.Impl;
214
215- system Critical_sel4
216-     features
217-         input: in event data port Base_Types::Integer;
218-         output: out event data port Base_Types::Integer;
219-     annex agree {**
220-         assume Req001_Critical_sel4 "Input is positive" : POSITIVE(input);
221-         guarantee Req002_Critical_sel4 "Output is positive" : POSITIVE(output);
222-     **};
223- end Critical_sel4;
224
225- system implementation Critical_sel4.Impl
226-     subcomponents
227-         Component_E: system Component_E_sel4.Impl;
228-         Component_A: process Component_A_sel4.Impl;
229-     connections
230-         c1: port input -> Component_A.input;
231-         c2: port Component_A.output -> Component_E.input;
232-         c3: port Component_E.output -> output;
233- end Critical_sel4.Impl;

```

Figure 48. Lines 176-191: sel4 process corresponding to thread Component_D; Lines 193-213: sel4 system corresponding to thread group Component_E; Lines 215-233: sel4 system corresponding to process Critical.

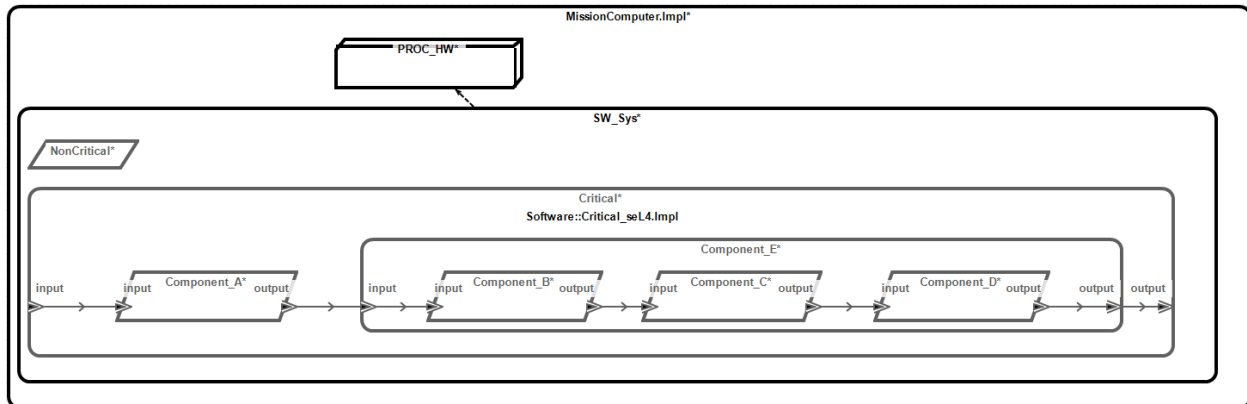


Figure 49. Transformed model.

Design Assurance

It is crucial to have evidence of design correctness both at the time of the model transformation is performed, and at any time up through system build. Resolute provides that assurance via augmentation of the requirement with assurance sub-claims as model transformations are performed.

When a requirement is imported from a TA1 tool, it will appear as in Figure 50.

```

4 goal Req_sel4(comp_context : component) <=
5   ** "[sel4] Software should run on a secure kernel" **
6   context Generated_By : "GearCASE";
7   context Generated_On : "Jul 20, 2020";
8   context Req_Component : "MissionComputer::MissionComputer.Impl.PROC_HW";
9   context Formalized : "False";
10  undeveloped

```

Figure 50. Imported requirement.

Initially, there is nothing for Resolute to check because the requirement hasn't yet been addressed in the design. Once the sel4 transform is applied, the requirement is updated with an additional rule to check, which reflects the sel4 transform action, as shown in Figure 51.

```

5 goal Req_sel4(comp_context : component, transformed_component : component) <=
6   ** "[sel4] Software should run on a secure kernel" **
7   context Generated_By : "GearCASE";
8   context Generated_On : "Jul 20, 2020";
9   context Req_Component : "MissionComputer::MissionComputer.Impl.PROC_HW";
10  context Formalized : "False";
11  sel4_transform(comp_context, transformed_component)

```

Figure 51. Modified requirement after sel4 transform.

The addition of the `sel4_transform()` call on line 11 provides Resolute with additional checks to make to ensure the requirement was addressed correctly. `sel4_transform()` is included in the `CASE_Model_Transformations` library and consists of one sub-claim:

- `each_thread_in_separate_process()` – Checks that each thread is contained in a separate process in the model for the target software system or process

To check whether the requirement has been correctly addressed in the design, select the containing component implementation (MissionComputer.Impl) and select Analyses → Resolute from the main menu. The Resolute output will appear in the output pane, as shown in Figure 52.

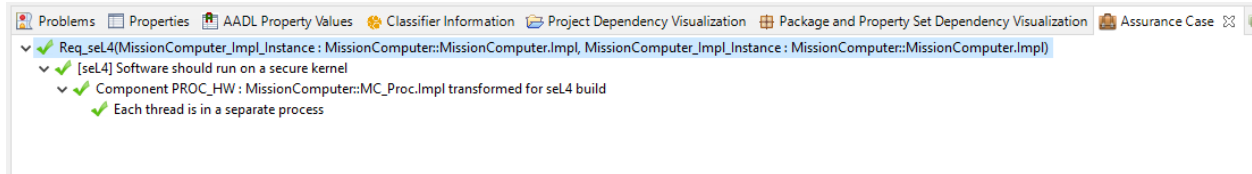


Figure 52. A passing Resolute analysis.

4. SPLAT

We have developed a tool called SPLAT (Semantic Properties for Language and Automata Theory) that supports the creation of, and correctness proofs for, filters specified by arithmetic properties extracted from AADL architectures. A record structure declaration with constraints on its fields specifies an encoding/decoding pair that maps a record of the given type into (and back out of) a sequence of bytes. A filter for such messages is also created, and embodied by a regular expression, which can be further compiled into a deterministic finite automata that checks that the sequence obeys the constraints. SPLAT extracts the filter properties from the architecture, creates encoders/decoders from the record declaration and accompanying constraints, creates a regular expression from the filter properties, and shows that the regular expression implements the filter properties. It produces a HOL theory capturing the formalization.

Before running SPLAT, the output directory needs to be set. To do so, select Window → Preferences from the main menu, and click the BriefCASE item on the left-hand side of the window (see Figure 53).

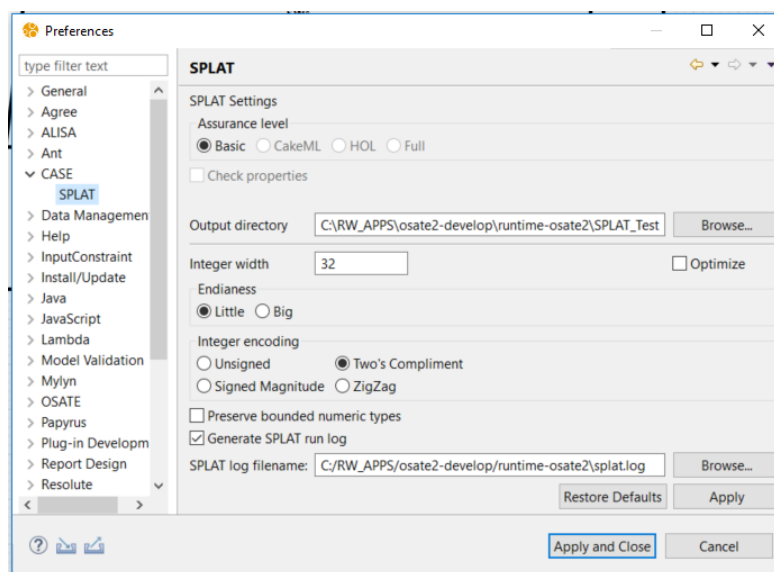


Figure 53. SPLAT preferences.

- SPLAT outputs go in the specified *Output directory*. This will include code, proof artifacts, and also files specifying the exact message formats. The outputs for each filter will go in a separate sub-directory of the specified output directory.
- SPLAT maintains a simple log of successful runs. This log is used by Resolute to determine whether SPLAT ran correctly. To enable this feature, select *Generate SPLAT run log*, and choose a location for the log file.

To run SPLAT, make sure the AADL package containing the top-level component implementation is active in the text editor, and select CASE → Cyber Resiliency → Synthesis Tools → SPLAT from the main menu. Note that currently, SPLAT can only be run on Linux. SPLAT will run in the background, but status information will appear in the console at the bottom of the OSATE environment. The line “Done with HOL proof of filter properties.” indicates that SPLAT has completed (see Figure 54).



```

Problems Properties AADL Property Values Classifier Information AGREE Results Assurance Case Console
HOL Proof of Filter Claims
Defined function "good_mission_window"
Defined function "good_command"
Defined function "WELL_FORMED_MESSAGE"
--> succeeded.
Constructing filters, encoders, decoders, and properties ... succeeded.
Proving filter properties ... succeeded.
45 states.
Generating code.
Finished.
Done with HOL proof of filter properties.

```

Figure 54. SPLAT status.

If SPLAT runs successfully, the specified output directory will contain four new files for each filter component in the model:

1. <Package Name>_<Filter Name>.c
2. <Package Name>Theory.dat
3. <Package Name>Theory.sig
4. <Package Name>Theory.sml

In addition, the *Source_Text* property of each filter component implementation in the AADL model will be updated with the location of the source file that SPLAT generates.

A test harness can be placed around the SPLAT-generated c file to input strings of a specific length. The output will be a true/false value that indicates whether the input string is well-formed or not, according to the filter specification.

Supported Constraints and Limitations

Splat translates well-formedness constraints on records into regular expressions. The records can be nested (records of records) but not recursive. Splat aims to handle records built from the following kinds of field:

- an element of an AADL base type (boolean or various flavor of integer only; floating point, chars, and strings are currently not supported, but are in the process of being added).
- an array (of fixed size)
- an element of an enumeration

- an element of a previously defined record

If a field is a number, then interval constraints are expected, where the interval endpoints are concrete integer literals. Arrays are expanded out, so constraints on array elements are uniformly applied. Enumerations: one can specify which of the enumerated constants are permitted.

A constraint may only apply to one field (e.g., a constraint that related two fields by saying they are equal is not supported). Also, constraints that aren't capturable by regular languages (e.g., that an array is sorted) are not going to work.

Example

Suppose we have declared the following types (in a generic notation)

```
coord = {lat : integer; lon : integer; alt : integer}
mode = Off | Running | Flying | TakingOff | Landing
```

and a record type

```
recd = {
    Auth : bool;
    Plan : coord [10];
    Mode : mode }
```

then the following example well-formedness predicate

```
wf_coord (cr) <=> (-90 <= cr.lat AND cr.lat <= 90) AND
    (-180 <= cr.lon AND cr.lon <= 180) AND
    (0 < cr.alt AND cr.alt < 15000)

wf_recd (r) <=> (r.Auth = true) AND
    (Forall c : r.Plan. wf_coord (c)) AND
    (r.Mode = Flying OR r.Mode = TakingOff)
```

will be translated to a regexp that checks binary-level data for conformance with the `wf_recd` predicate.

Numeric field constraints that are inequalities {<, <=, >, >=} over record fields and integer literals are supported. Also supported is the specification of a subset of an enumeration via a disjunction setting forth the allowable constructors.

```
<constr> ::= <ilit> <ineq> <recd-proj>
          | <recd-proj> <ineq> <ilit>
          | (<recd-proj> '=' <id> OR ... OR <recd-proj> '=' <id>)
```

```
<ineq> ::= '<' | '<=' | '>' | '>='  
<ilit> ::= ('-')?<digit>+  
  
<recd-proj> ::= <qid>.fldname   ;;; record projection of form A.field  
<qid> ::= (<qid>'.')*id
```

Currently, SPLAT does not support floating point data types. Floating point support will be addressed in future versions. In addition, the following run options (specified in SPLAT preferences) are still under construction, and therefore not currently supported:

- Assurance Level: CakeML, HOL, Full
- Check properties

5. HAMR

HAMR ("[H]igh [A]ssurance [M]odeling and [R]apid engineering for embedded systems") is a code generation and system build framework for cyber-resilient systems whose architecture is using the Architecture and Analysis Definition Language (AADL). HAMR supports development of new components and wrapping of legacy components by generating C code that provides interfacing infrastructure between components. Once component implementations are developed, HAMR can create deployable builds that are linked to assurance artifacts produced by other DARPA CASE tools.

For interface and infrastructure generation, HAMR translates system models in AADL to C code that implements threading infrastructure and inter-component communication for AADL components. HAMR does not generate code for the application logic of a system; it only generates infrastructure code for "gluing together" the component application code. So to "program a system" using HAMR, you define a component-based system/software architecture in AADL, program the internal behavior (the "application code") of the components in a programming language (HAMR supports several different languages, plus some high-level declaration approaches for generating message filters for cyber-resiliency) and then HAMR generates the infrastructure code (aligned with the AADL architecture) that provides an execution context for the application code.

The auto-generated infrastructure code is designed to implement the semantics of AADL threading and communication as specified in the AADL standard (HAMR implements a *subset* of the standard semantics relevant for CASE). In essence, AADL defines a *computational model* -- a constrained way of organizing processes, threads, and communication -- that matches the domain properties of real-time embedded systems. AADL and its computational model are system engineering abstractions that are designed to be *analyzable*. Since the HAMR auto-generated infrastructure code is faithful to the AADL semantics and computational model, various forms of analysis applied to AADL such as the cyber-resiliency analyses being developed in DARPA CASE are faithful to the semantics and resiliency properties of the executable code in deployed system.

The application code (i.e., business logic) that provides the functionality of AADL components is written in one of several languages (including C, CakeML -- a language with verified machine code generation and accompanying proof tools that enable applications to be proved correct using theorem-proving technology, or Slang -- a safety-critical subset of Scala). The business logic and associated executable code is held in a location known to HAMR. HAMR takes the user-written application code and weaves it together with the auto-generated infrastructure code to form a system build.

The seL4 verified micro-kernel is a key technology solution in DARPA CASE that is used to provide strong partitioning between components in an application. With this partitioning, one application component cannot access the memory/state of another component unless granted that capability, and communication between components is guaranteed to be limited to explicitly declared pathways. Based on these partitioning properties, seL4 plays a key role in achieving cyber-resiliency by

- isolating less trusted portions of the application in separate components so that they cannot attack or interfere with critical components,
- separating trusted components to better isolate faults and achieve other robustness properties, and
- enabling communication between components to be more easily assessed/audited and guarded with various security controls.

HAMR can generate infrastructure code for several platforms, such as seL4 and Linux (either stand-alone Linux applications, or virtual machines on seL4 that host Linux applications). When HAMR translates to seL4, it uses seL4 partitioning facilities to create separate partitions for AADL components/subsystems and seL4 inter-partition communication to realize AADL inter-component communication. To realize the translation to seL4, HAMR translates AADL component interfaces and connection topologies to CAMkES -- a dedicated language for specifying seL4 components and inter-component communication. HAMR invokes a Data61-provided translation tool that translates CAMkES to C and seL4 capability configuration files. Other HAMR-supported platforms such as Linux do not provide strong partitioning. Nevertheless, HAMR generates code for these platforms in a manner that organizes the system build into distinct units with constrained interactions between each unit. Even without the strong OS/platform partitioning, this disciplined build structure can aid assurance arguments. Moreover, HAMR's support for Linux and Linux in VMs facilitates wrapping of legacy components and subsystems (perhaps untrusted) and isolating them in a VM in a seL4 partition.

An important aspect of the HAMR tool chain is that it enables the application code to be platform agnostic to a significant degree. Application programmers program to a common collection of APIs for threading and communication aspects (corresponding to the AADL computation model) that hides the details of how these aspects are realized on particular platform. Use of this abstraction makes it easier to move implementations between platforms (e.g., shifting code running in a Linux process to code running on bare-metal in a seL4 CAMkES component -- as might be necessary when extracting key components out of an untrusted Linux-based deployment and hardening them to trusted components to run in an seL4 partition). Note that platform specific I/O interfacing to sensors/actuators, etc. may still need to be reworked when moving to another platform.

Overall, HAMR forms the backbone of tool-chain for engineering cyber-resilient embedded systems by supporting modeling, analyzing, transforming, code generation, and build support for partitioned applications.

HAMR Translation Architecture

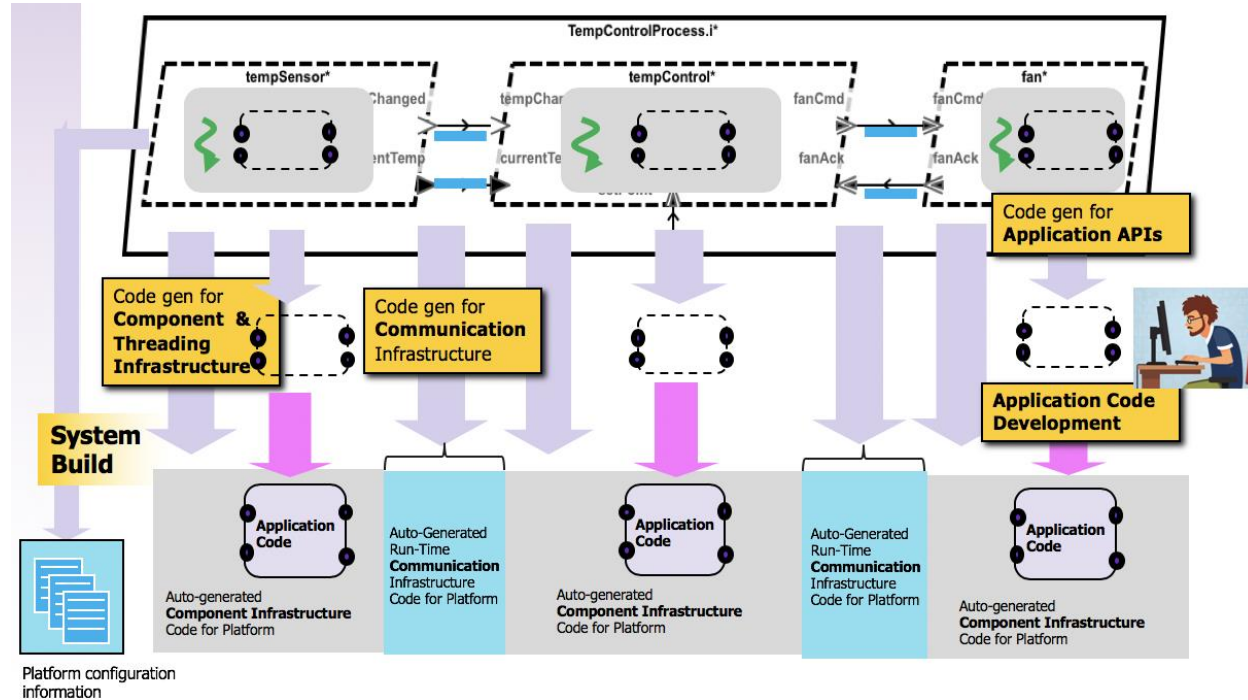


Figure 55. HAMR Translation Architecture.

Figure 55 presents the high-level concept of the HAMR code generation. The AADL model in the figure is a simple temperature control system with a temperature sensor, a controller (thermostat), and a cooling fan – the details are not relevant for this overview.

A fundamental goal of HAMR is to structure the code so that the application code is isolated from the underlying component and communication infrastructure – enabling HAMR to target platforms with different threading and communication with minimal changes to application code. To achieve this, the code generation has the following four dimensions.

- Code skeletons and APIs for application code – for each AADL thread, HAMR generates code skeletons for application code (see right hand side of Figure 55 – forthcoming DARPA CASE releases will document the details of these code skeletons). The developer completes the component implementation using a conventional development environment. The application code accesses automatically generated APIs to communicate via the component's ports with other components. These APIs abstract the application code away from the details of the particular communication infrastructure.
- Component infrastructure, including threading infrastructure – code is generated to support the invocation of the application code and move the data being transported over the component's ports between port variables accessible by the application code and the communication infrastructure abstracted by the communication APIs.

- Communication infrastructure – code is generated to move data/events between output ports of sending components and input ports of receiving components. The particular mechanism used depends on the underlying platform. For example, for a Unix/Linux platform-based implementation, System V communication primitives are used. For an seL4-based implementation, seL4's inter-partition communication primitives are used. For distributed components, a publish-subscribe framework like OMG Data Distribution Service or a CAN BUS could be used (distributed communication is not yet implemented in HAMR).
- Platform configuration information - Depending on how the underlying platform capabilities are configured, HAMR may generate meta-data (e.g., in an XML or JSON format) that is used to configure options on the underlying platform.

Overall, HAMR can be seen as coordinating an AADL-guided build of a system that includes the four dimensions above. The overall assurance case that the generated build achieves desired system functional/safety/security requirements (including cyber-resiliency requirements) includes the following elements:

- (Arch) arguments that an architecture appropriate for the system requirements (e.g., achieving appropriate partitioning properties) is specified in AADL (reflected in the topological structure of the AADL specification),
- (ArchAttributes) arguments that system attributes presented (e.g. as AADL properties) are appropriate for the system requirements,
- (ArchAnalysis) arguments that any automated analyses including information flow analysis, latency analysis, schedulability analysis, etc., that are used to automatically configure model attributes or to confirm that model attributes appropriately realize system requirements,
- (ApplicationCode) arguments each component's application code is implemented to satisfy component behavioral specification so that integration of the components with component/communication infrastructure that adheres to the AADL computational model yields "end to end" behavior that achieves the system requirements,
- (ComponentInfrastructureCode) arguments that the HAMR code generation for component infrastructure correctly implements the AADL computational model,
- (CommunicationInfrastructureCode) arguments that the HAMR code generation for communication infrastructure correctly implements the AADL computational model,
- (PlatformConfigurationInformation) arguments that the HAMR code generation for platform configuration information configures the platform to achieve resource provisioning/partitioning and other properties necessary to realize the system requirements.

Examples

Several example projects are provided that document the CAMkES code generated by HAMR for the various types of AADL port connections, which can be found here:

<https://github.com/loonwerks/CASE/tree/master/TA5/tool-evaluation-3/HAMR/examples>

The documentation for each project also includes instructions on how to run HAMR on the corresponding AADL model in order to generate CAMkES code.

6. BriefCASE-Compatible Tools

AGREE

The Assume Guarantee REasoning Environment (AGREE) is a compositional, assume-guarantee-style model checker for AADL models. It is compositional in that it attempts to prove properties about one layer of the architecture using properties allocated to subcomponents. The composition is performed in terms of assumptions and guarantees that are provided for each component. Assumptions describe the expectations the component has on the environment, while guarantees describe bounds on the behavior of the component. AGREE uses k-induction as the underlying algorithm for the model checking.

The main idea is that complex systems are likely to be designed as a hierarchical federation of systems. As we descend the hierarchy, design information at some level turns into requirements for subsystems at the next lower level of abstraction. These hierarchical levels can be straightforwardly expressed in AADL. What we would like to support, therefore, is:

- An approach to requirements validation, architectural design, and architectural verification that uses the requirements to drive the architectural decomposition and the architecture to iteratively validate the requirements, and
- An approach to verify and validate components prior to building code-level implementations

AGREE is a first step towards realizing this vision. Components and their connections are specified using AADL and annotated with assumptions that components make about the environment and guarantees that the components will make about their outputs if the assumptions are met. Each layer of the system hierarchy is verified individually; AGREE attempts to prove the system-level guarantees in terms of the guarantees of its components.

AGREE is currently included in the Collins CASE toolchain. The source code can be found here:

<https://github.com/loonwerks/formal-methods-workbench/tree/master/tools/agree>

The AGREE User's Guide can be found here:

<https://github.com/loonwerks/formal-methods-workbench/tree/master/documentation/agree>

Resolute

Arguments about the safety, security, and correctness of a complex system are often made in the form of an assurance case. An assurance case is a structured argument, often represented with a graphical interface that presents and supports claims about a system's behavior. The argument may combine different kinds of evidence to justify its top level claim. While assurance cases deliver some level of guarantee of a system's correctness, they lack the rigor that proofs from formal methods typically provide. Furthermore, changes in the structure of a model during development may result in inconsistencies between a design and its assurance case. Our solution is a framework for automatically generating assurance cases based on 1) a system model specified in an architectural design language, 2) a set of logical rules expressed in a domain specific language that we have developed, and 3) the results of other formal analyses that have been run on the model. The rigor of these automatically generated

assurance cases exceeds those of traditional assurance case arguments because of their more formal logical foundation and direct connection to the architectural model.

Resolute is currently included in the Collins CASE toolchain. The source code can be found here:

<https://github.com/loonwerks/formal-methods-workbench/tree/master/tools/resolute>

The Resolute User's Guide can be found here:

https://github.com/loonwerks/formal-methods-workbench/blob/master/documentation/resolute/user_guide.pdf

Resolint

Resolint is an OSATE-based linter tool for AADL models. Resolint provides a language for specifying rules that correspond to modeling guidelines, as well as a checker for evaluating whether a model complies with the rules. Results of the Resolint analysis are displayed to the user. Rule violations will indicate severity, and are linked to the model element that is out of compliance with the rule.

Resolint is currently included with the Resolute plugin for OSATE (and bundled into the Collins CASE toolchain). The source code can be found here:

<https://github.com/loonwerks/formal-methods-workbench/tree/master/tools/resolute>

The Resolint User's Guide can be found here:

https://github.com/loonwerks/formal-methods-workbench/blob/master/documentation/resolute/Resolint_Users_Guide.pdf

7. AADL Modeling Guidelines

AADL has been engineered as a general-purpose system architecture modeling language. As a result, the language specification does not necessarily dictate the semantics of how the modeling artifacts in AADL are mapped to actual physical artifacts in the end systems. The way in which AADL-based analysis and code-generation tools interpret the language's modeling artifacts are domain specific, and are left to the tool developers.

A set of modeling guidelines for producing well-formed AADL models is therefore required to specify the subset of well-formed AADL that the Collins CASE toolchain will accept. In addition, a mechanism for checking compliance with the guidelines will greatly aid model development.

BriefCASE includes both a set of modeling guidelines and a tool for checking compliance. The modeling guidelines document, "AADL Modeling Guidelines for CASE" can be found in the documentation directory of this release. Compliance with the guidelines can be accomplished using the Resolint tool, described in the previous section.