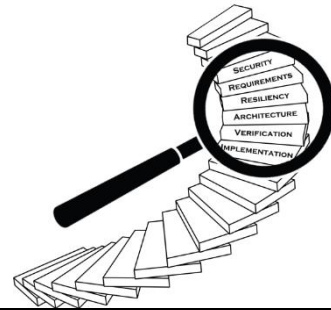


# Collins StairCASE User's Guide

December 2019

## StairCASE: Secure Transformation of Architecture, Implementation, and Requirements for Cyber Assured Systems Engineering



### Contents

Introduction .....	2
1. Generating and Importing Cyber Requirements.....	2
2. Model Annotations .....	8
3. Model Transformations .....	9
Filter .....	9
Attestation .....	15
Virtualization.....	21
4. SPLAT.....	25
5. HAMR.....	29
6. StairCASE-Compatible Tools .....	33
AGREE.....	33
Resolute .....	33
Resolint .....	34
7. AADL Modeling Guidelines .....	34

## 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

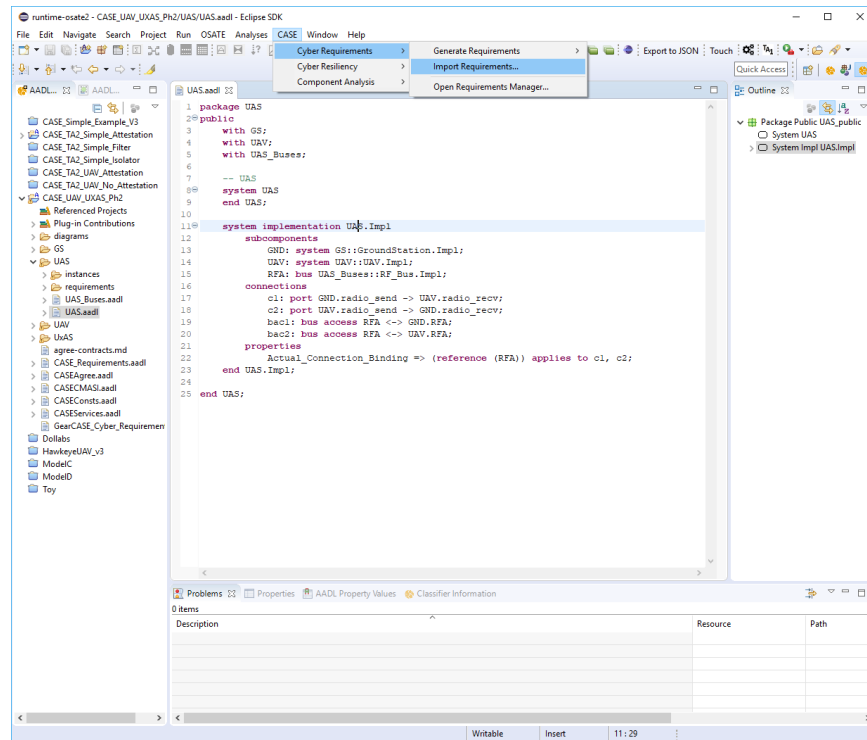
StairCASE interfaces with tools to generate cyber-security requirements for AADL models to improve their cyber-resiliency. There are multiple steps in this process. In this section, we describe the following steps:

1. Running a cyber requirements tool from within the CASE environment.
2. Importing 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 OSATE menubar. 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 CASE tools and will need to be installed separately. Future versions of the Collins CASE environment will include them.

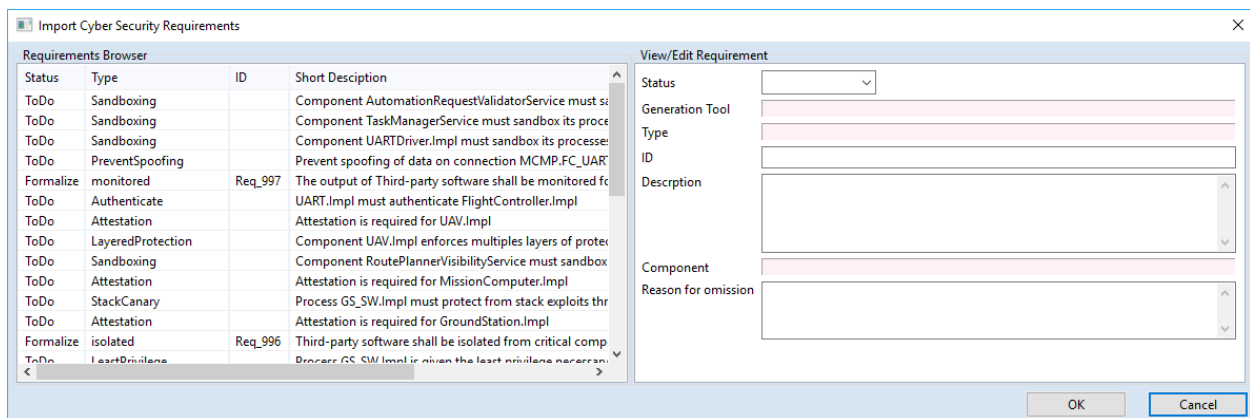
#### Invoking the Cyber Requirements Tool

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 via the OSATE menu (e.g., CASE → Cyber Requirements → Generate Requirements → GearCase).

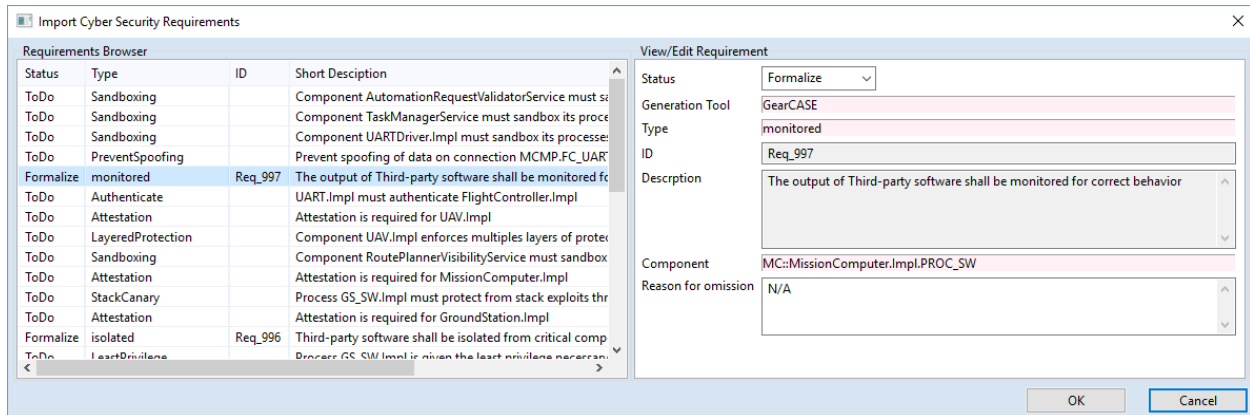


## Import Requirements Dialog

When the cyber requirements tools are integrated into the CASE toolchain, the generated cyber-security requirements are automatically displayed to the user in the StairCASE Import Requirements dialog.



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

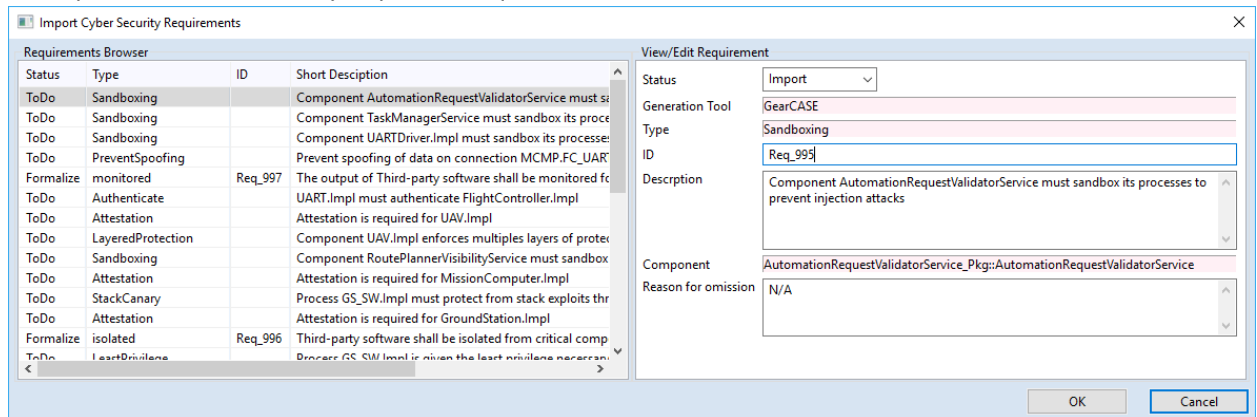


Status	Type	ID	Short Description
ToDo	Sandboxing		Component AutomationRequestValidatorService must s
ToDo	Sandboxing		Component TaskManagerService must sandbox its proce
ToDo	Sandboxing		Component UARTDriver.Impl must sandbox its processe
ToDo	PreventSpoofing		Prevent spoofing of data on connection MCMP.FC_UAR
Formalize	monitored	Req_997	The output of Third-party software shall be monitored fo
ToDo	Authenticate		UART.Impl must authenticate FlightController.Impl
ToDo	Attestation		Attestation is required for UAV.Impl
ToDo	LayeredProtection		Component UAV.Impl enforces multiples layers of prote
ToDo	Sandboxing		Component RoutePlanner/VisibilityService must sandbox
ToDo	Attestation		Attestation is required for MissionComputer.Impl
ToDo	StackCanary		Process GS_SW.Impl must protect from stack exploits thr
ToDo	Attestation		Attestation is required for GroundStation.Impl
Formalize	isolated	Req_996	Third-party software shall be isolated from critical comp
ToDo	LeastPrivilege		Process GS_SW.Impl is given the least privilege neces

**View/Edit Requirement**  
 Status: Formalize  
 Generation Tool: GearCASE  
 Type: monitored  
 ID: Req\_997  
 Description: The output of Third-party software shall be monitored for correct behavior  
 Component: MC::MissionComputer.Impl.PROC\_SW  
 Reason for omission: N/A

Through this interface, the user can perform the following tasks:

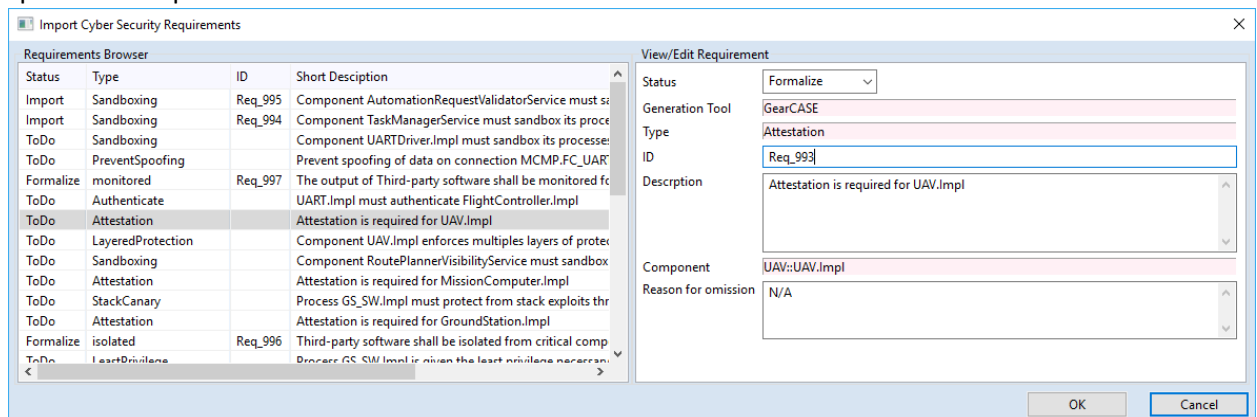
1. **Import Requirements:** Adds skeletal Resolute clauses to the AADL model. The user must provide a unique identifier for every imported requirement.



Status	Type	ID	Short Description
ToDo	Sandboxing		Component AutomationRequestValidatorService must s
ToDo	Sandboxing		Component TaskManagerService must sandbox its proce
ToDo	Sandboxing		Component UARTDriver.Impl must sandbox its processe
ToDo	PreventSpoofing		Prevent spoofing of data on connection MCMP.FC_UAR
Formalize	monitored	Req_997	The output of Third-party software shall be monitored fo
ToDo	Authenticate		UART.Impl must authenticate FlightController.Impl
ToDo	Attestation		Attestation is required for UAV.Impl
ToDo	LayeredProtection		Component UAV.Impl enforces multiples layers of prote
ToDo	Sandboxing		Component RoutePlanner/VisibilityService must sandbox
ToDo	Attestation		Attestation is required for MissionComputer.Impl
ToDo	StackCanary		Process GS_SW.Impl must protect from stack exploits thr
ToDo	Attestation		Attestation is required for GroundStation.Impl
Formalize	isolated	Req_996	Third-party software shall be isolated from critical comp
ToDo	LeastPrivilege		Process GS_SW.Impl is given the least privilege neces

**View/Edit Requirement**  
 Status: Import  
 Generation Tool: GearCASE  
 Type: Sandboxing  
 ID: Req\_995  
 Description: Component AutomationRequestValidatorService must sandbox its processes to prevent injection attacks  
 Component: AutomationRequestValidatorService\_Pkg::AutomationRequestValidatorService  
 Reason for omission: N/A

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

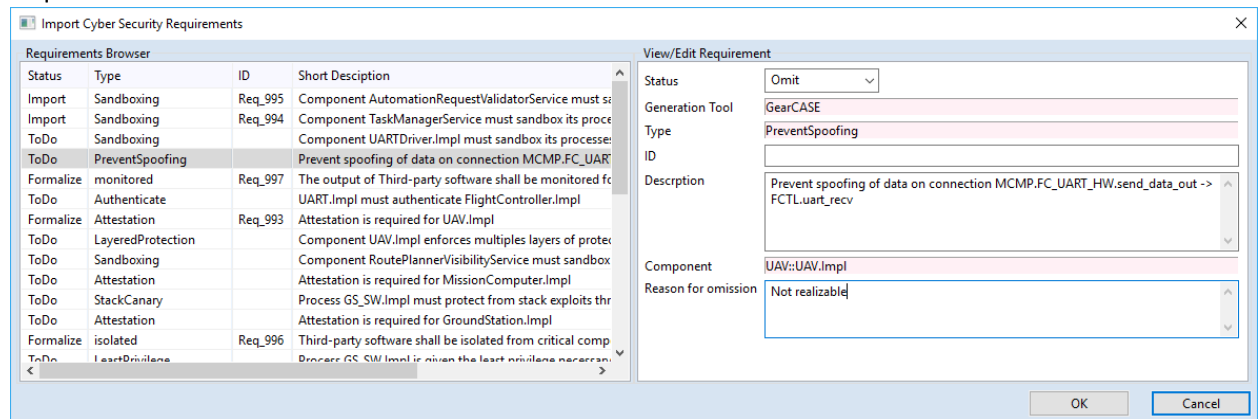


Status	Type	ID	Short Description
Import	Sandboxing	Req_995	Component AutomationRequestValidatorService must s
Import	Sandboxing	Req_994	Component TaskManagerService must sandbox its proce
ToDo	Sandboxing		Component UARTDriver.Impl must sandbox its processe
ToDo	PreventSpoofing		Prevent spoofing of data on connection MCMP.FC_UAR
Formalize	monitored	Req_997	The output of Third-party software shall be monitored fo
ToDo	Authenticate		UART.Impl must authenticate FlightController.Impl
ToDo	Attestation		Attestation is required for UAV.Impl
ToDo	LayeredProtection		Component UAV.Impl enforces multiples layers of prote
ToDo	Sandboxing		Component RoutePlanner/VisibilityService must sandbox
ToDo	Attestation		Attestation is required for MissionComputer.Impl
ToDo	StackCanary		Process GS_SW.Impl must protect from stack exploits thr
ToDo	Attestation		Attestation is required for GroundStation.Impl
Formalize	isolated	Req_996	Third-party software shall be isolated from critical comp
ToDo	LeastPrivilege		Process GS_SW.Impl is given the least privilege neces

**View/Edit Requirement**  
 Status: Formalize  
 Generation Tool: GearCASE  
 Type: Attestation  
 ID: Req\_993  
 Description: Attestation is required for UAV.Impl  
 Component: UAV::UAV.Impl  
 Reason for omission: N/A

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 a reason for omitting the

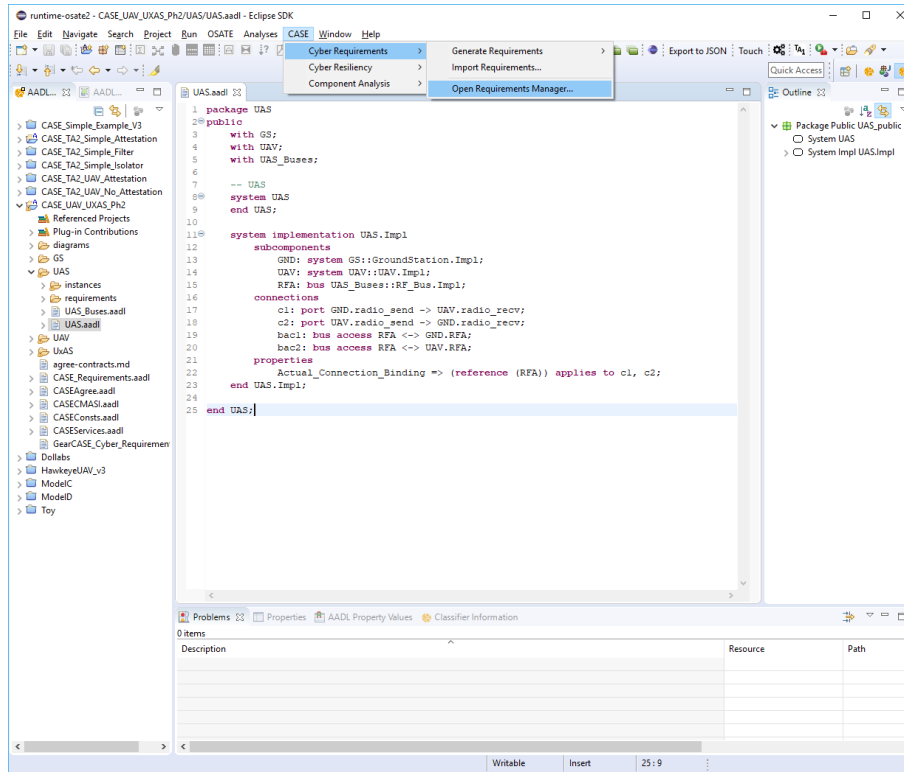
requirement.



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.

## Requirements Manager

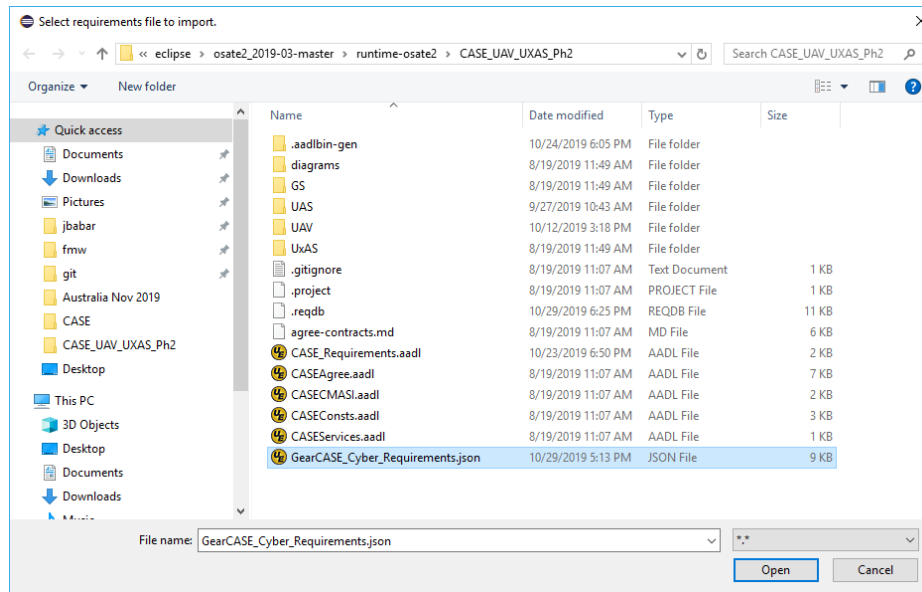
Imported requirements can also be modified in the Requirements Manager (invoked via CASE → Cyber Requirements → Generate Requirements → Open Requirements Manager...). This allows the user to import, omit, or modify requirements in the model over multiple sessions. It uses the same interface as the Import 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.



Once the user is satisfied with the state of the AADL model with respect to the cyber-security requirements, the cyber requirements tool can be called again to analyze vulnerabilities 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.

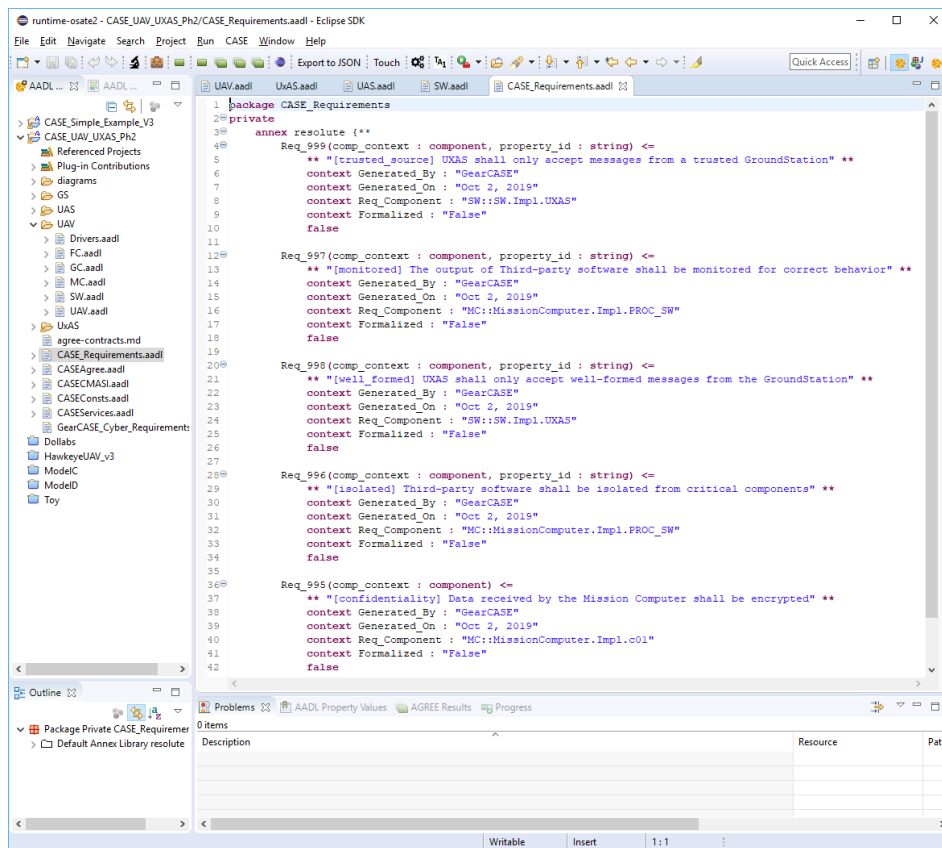
### Importing requirements from a file

The Import Requirements dialog can also be invoked via the OSATE menu (CASE → Cyber Requirements → Import Requirements...). This brings up a file dialog to select the requirements file to be imported. This can be useful when working with cyber requirements tools that have not been fully integrated into the CASE toolchain.



## Notes

1. The file CASE\_Requirements.aadl is added to the root folder 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 OSATE 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 is usually a 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. StairCASE includes an AADL CASE property set, in addition to a user interface for simplifying the annotation of AADL components.

The CASE\_Properties property set is provided with StairCASE, and is accessible in an AADL project in OSATE in the Plugin Contributions folder, as shown in the OSATE Navigation pane in Figure 1.

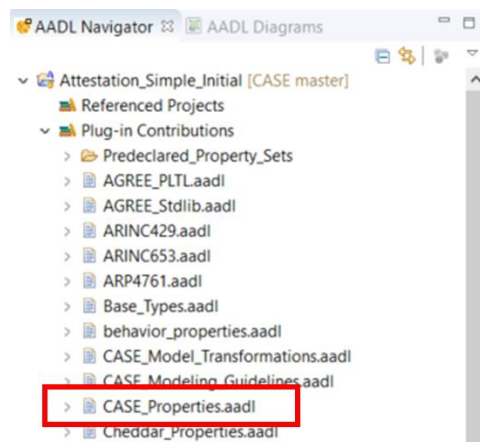


Figure 1. CASE\_Properties property set



To annotate an AADL element with a specific CASE property, select the element in the text editor and click the CASE → Cyber Resiliency → Model Annotations... menu option. A window will open, similar to that pictured in Figure 2. 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.

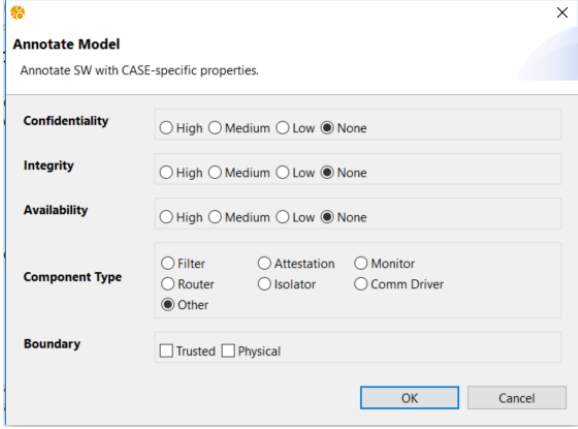


Figure 2. Model Annotations.

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

StairCASE 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](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/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.

The Filter transform can also be performed on the CASE Phase 1 UAV example model. Three versions of the model are available for reference:

- Initial model – This is the Phase 1 UAV model that includes an imported cyber requirement, which drives the well-formedness of incoming messages to the FlightPlanner component. The Initial model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Filter/UAV\\_Example/Initial\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Filter/UAV_Example/Initial_Model)
- Transformed model – This is the Phase 1 UAV model after the Filter transform has been applied. The Transformed model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Filter/UAV\\_Example/Transformed\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Filter/UAV_Example/Transformed_Model)
- Test model – This is the Phase 1 UAV model containing several software implementations for testing the correctness of the Resolute evaluation on the Filter transform. The Test model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Filter/UAV\\_Example/Test\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Filter/UAV_Example/Test_Model)

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 3 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.

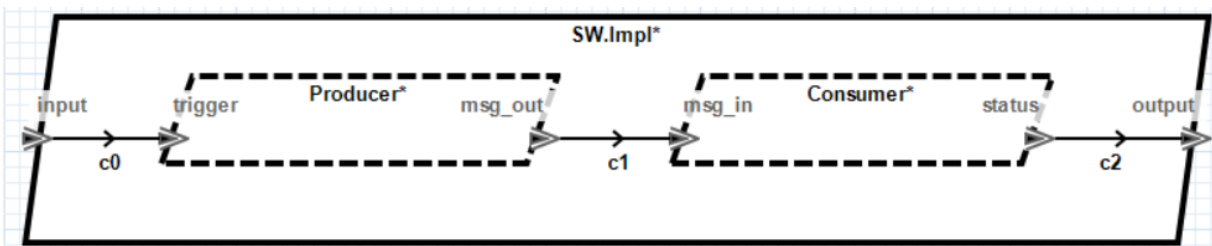


Figure 3. Initial model.

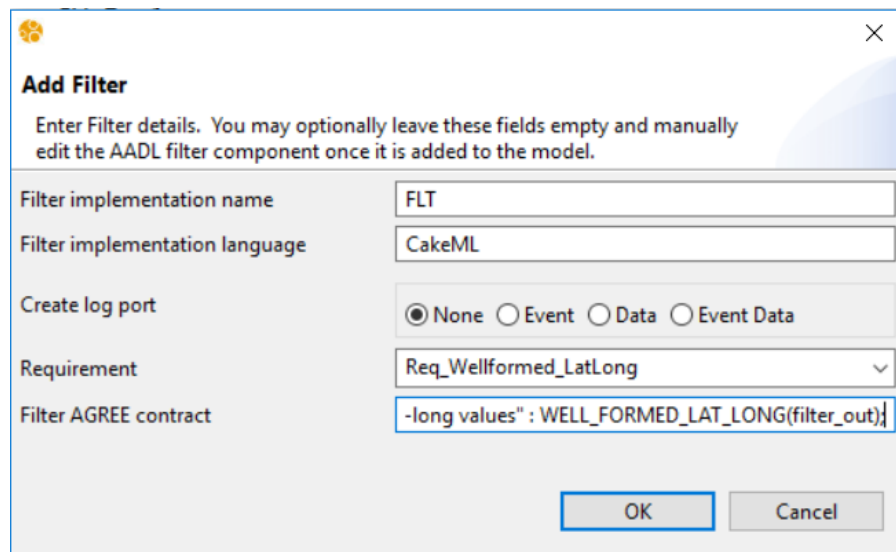
A filter can be added to the following AADL components:

- Thread
- Thread Group
- Process containing a single thread – Not yet implemented

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). Also note that AADL feature groups are not currently supported, but will be in future versions of the tool.

To insert a filter, select the connection in a component implementation that terminates at the component that requires filtered input (for example, in `Producer_Consumer.aadl`, select the `c1` connection on line 55). 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 menubar, click the `CASE` → `Cyber Resiliency` → `Model Transformations` → `Add Filter...` menu item. A wizard will open, as shown in Figure 4. The wizard enables the user to customize the filter, including providing the filter specification.



The 'Add Filter' wizard dialog box contains the following fields and controls:

- Filter implementation name:** Text input field with the value 'FLT'.
- Filter implementation language:** Text input field with the value 'CakeML'.
- Create log port:** Radio button group with options: ☒ None, ☐ Event, ☐ Data, ☐ Event Data.
- Requirement:** Dropdown menu with the selected value 'Req\_Wellformed\_LatLong'.
- Filter AGREE contract:** Text input field with the value '-long values" : WELL\_FORMED\_LAT\_LONG(filter\_out);'.
- Buttons:** 'OK' and 'Cancel' buttons at the bottom right.

Figure 4. Add Filter wizard

The Filter transform will create a special `CASE_Filter` AADL component type and implementation, and insert them into the model. It will then instantiate the `CASE_Filter` as a subcomponent in the selected implementation. The user may provide a name for the filter subcomponent, 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.

By default the CASE filter will drop any messages that do not match the regular expression 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.

The requirement drop-down box lists all of the cyber-requirements that have been imported from TA1 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 filter specification as an AGREE *guarantee* statement. This is typically done by referring to the outgoing message on the filter's output port. The message type will be the same as the target component's input port. Within the AGREE statement, the filter output port name by *filter\_out*. For example, if the message type is a signed integer, and the filter should drop any message with a value less than zero, the AGREE statement will be:

```
guarantee Req001_Filter "Only non-negative numbers" : filter_out >= 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 79), and so the filter AGREE statement will be:

```
guarantee Req_Filter_LatLong "The Consumer shall only receive well-formed lat-long values" : WELL_FORMED_LAT_LONG(filter_out);
```

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

Clicking the OK button on the wizard will insert the `CASE_Filter` into the model, as shown in Figure 5 and Figure 6. The graphical representation is shown in Figure 7.

```

9=  thread CASE_Filter
10      features
11          filter_in: in event data port Coordinate.Impl;
12          filter_out: out event data port Coordinate.Impl;
13      properties
14          CASE_Properties::COMP_TYPE => FILTER;
15          CASE_Properties::COMP_SPEC => ("Req_Filter_LatLong");
16=  annex agree {**
17      guarantee Req_Filter_LatLong "The Consumer shall only receive well-formed lat-long values" : WELL_FORMED_LAT_LONG(filter_out);
18      **};
19  end CASE_Filter;
20
21=  thread implementation CASE_Filter.Impl
22  end CASE_Filter.Impl;
```

Figure 5. Line 9: `CASE_Filter` component type; Line 21: `CASE_Filter` component implementation.

```

68=  process implementation SW.Impl
69      subcomponents
70          Producer: thread Producer.Impl;
71          Consumer: thread Consumer.Impl;
72          FLT: thread CASE_Filter.Impl;
73      connections
74          c0: port input -> Producer.trigger;
75          c1: port Producer.msg_out -> FLT.filter_in;
76          c3: port FLT.filter_out -> Consumer.msg_in;
77          c2: port Consumer.status -> output;
78=  annex resolute {**
79      prove Req_Wellformed_Alt(this.Consumer, "Req_Wellformed_Alt")
80      prove Req_Wellformed_LatLong(this.Consumer, "Req_Wellformed_LatLong", this.FLT, this.c3, Coordinate.Impl)
81      **};
82  end SW.Impl;
```

Figure 6. Line 72: filter subcomponent; Lines 75-76: filter connections; Line 80: updated assurance claim call.

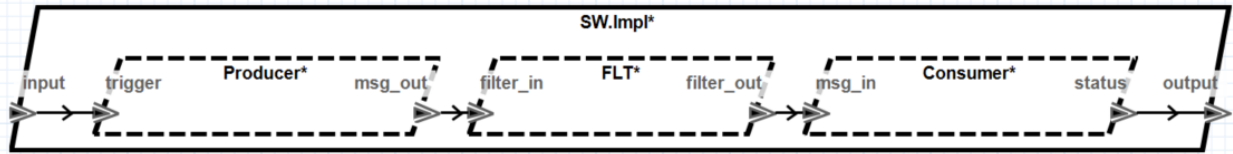


Figure 7. Transformed model containing a filter.

## Compound Filters

Two CASE filters cannot be connected to each other. Attempting to place a filter on a connection that already has a filter component as either its source or destination will result in a warning, and present an option to create a compound filter, as shown in Figure 8. A compound filter is simply a single filter component containing multiple filter expressions.

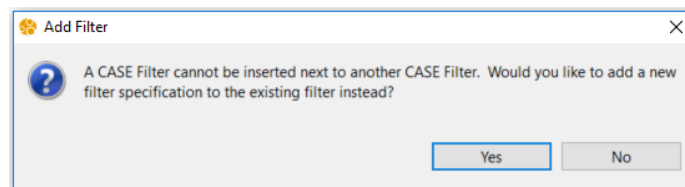


Figure 8. Adjacent CASE Filter warning.

Selecting “Yes” will display a wizard similar to inserting a new filter, however, some fields will be disabled since only a new filter expression is being added (see Figure 9). The result of creating a compound filter is an additional AGREE guarantee statement in the filter component. The conjunction of these specifications describe the filter property.

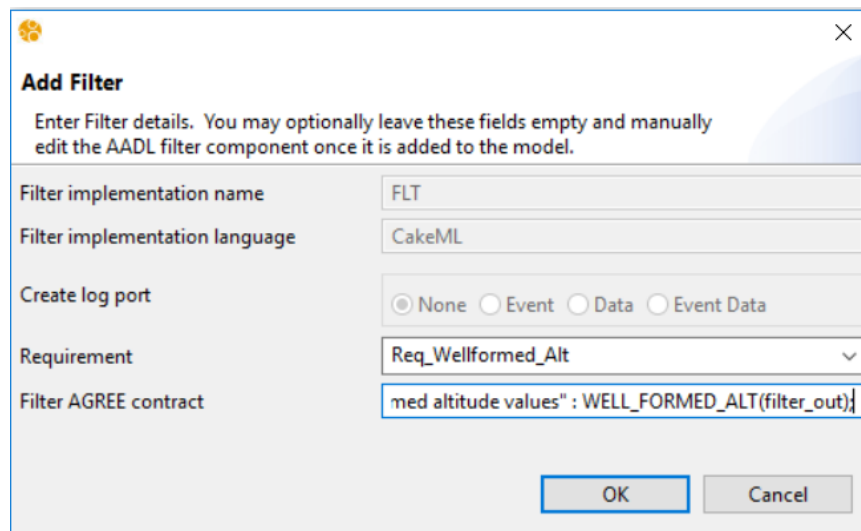


Figure 9. Adding an expression to an existing filter.

For example, by adding the additional filter expression:

```
guarantee Req_Filter_Alt "The Consumer shall only receive well-formed altitude values" : WELL_FORMED_ALT(filter_out);
```

the new expression will appear in the existing filter as shown in Figure 10.

```

9@  thread CASE_Filter
10      features
11          filter_in: in event data port Coordinate.Impl;
12          filter_out: out event data port Coordinate.Impl;
13      properties
14          CASE_Properties::COMP_TYPE => FILTER;
15          CASE_Properties::COMP_SPEC => ("Req_Filter_LatLong", "Req_Filter_Alt");
16@  annex agree {**
17      guarantee Req_Filter_LatLong "The Consumer shall only receive well-formed lat-long values" : WELL_FORMED_LAT_LONG(filter_out);
18      guarantee Req_Filter_Alt "The Consumer shall only receive well-formed altitude values" : WELL_FORMED_ALT(filter_out);
19      **};
20  end CASE_Filter;
21
22@  thread implementation CASE_Filter.Impl
23  end CASE_Filter.Impl;

```

Figure 10. Compound filter.

## CASE Filter Properties

For filter synthesis using SPLAT, two other properties are necessary. The `CASE_Properties::COMP_TYPE => FILTER` property association indicates that the component is a CASE Filter. The `CASE_Properties::COMP_SPEC` property association lists the AGREE specification IDs of the guarantee statements that comprise the filter expression. For example, in Figure 10, the `COMP_SPEC` property lists two identifiers corresponding to the two AGREE guarantee statements on lines 17 and 18. The specification of these two statements, `WELL_FORMED_LAT_LONG()` and `WELL_FORMED_ALT()`, provide 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 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 subclaims as model transformations are performed.

When a requirement is imported from a TA1 tool it will appear as in Figure 11.

```

72@  Req_Wellformed_LatLong(comp_context : component, property_id : string) <=
73      ** "[well_formed] The Consumer shall only receive well-formed latitude-longitude coordinates" **
74      agree_prop_checked(comp_context, property_id)

```

Figure 11. 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 12.

```

100= Req_Wellformed_LatLong(comp_context : component, property_id : string, filter : component, conn : connection, message_type : data) <=
101   ** "[well_formed] The Consumer shall only receive well-formed latitude-longitude coordinates" **
102   agree_prop_checked(comp_context, property_id) and add_filter(comp_context, filter, conn, message_type)

```

Figure 12. Modified requirement after Filter transform.

The addition of the `add_filter()` call on line 102 provides Resolute with additional checks to make to ensure the requirement was addressed correctly. In this case, `add_filter()` is included in the `CASE_Model_Transformations` library 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
- `filter_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 menubar. The Resolute output will appear in the output pane, as shown in Figure 13.

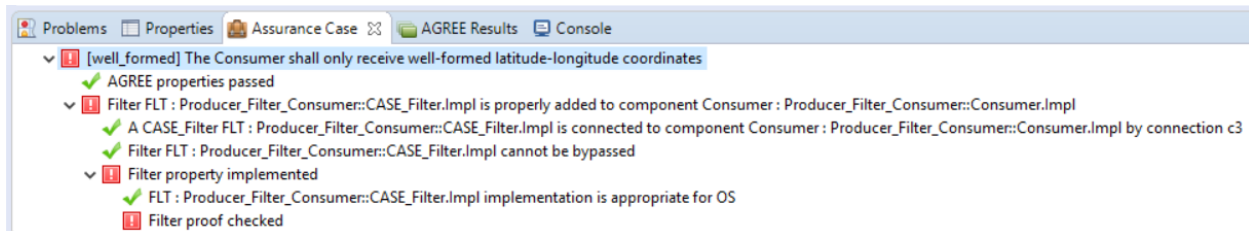


Figure 13. A failing Resolute analysis.

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 14.

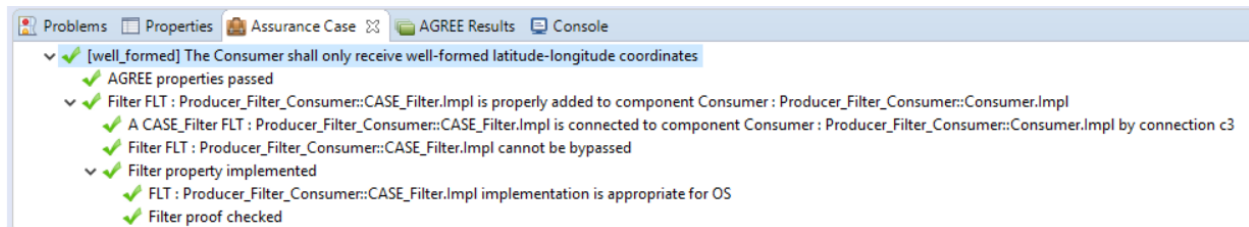


Figure 14. A passing Resolute analysis.

## 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](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.

The Attestation transform can also be performed on the CASE Phase 1 UAV example model. Three versions of the model are available for reference:

- Initial model – This is the Phase 1 UAV model that includes an imported cyber requirement, which drives the attestation of the source of incoming messages to the FlightPlanner component. The Initial model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model Transformations/Attestation/UAV Example/Initial Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/Attestation/UAV%20Example/Initial%20Model)
- Transformed model – This is the Phase 1 UAV model after the Attestation transform has been applied. The Transformed model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model Transformations/Attestation/UAV Example/Transformed Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/Attestation/UAV%20Example/Transformed%20Model)
- Test model – This is the Phase 1 UAV model containing several software implementations for testing the correctness of the Resolute evaluation on the Attestation transform. The Test model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model Transformations/Attestation/UAV Example/Test Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model%20Transformations/Attestation/UAV%20Example/Test%20Model)

A CASE Attestation Manager is added to a communication driver component to ensure that only messages from trustworthy sources are accepted. Only components with the CASE\_Properties::COMP\_TYPE = COMM\_DRIVER property association can have an Attestation Manager connected to it, and only one Attestation Manager may be connected to a communication driver. All outgoing connections from the communication driver will pass through the Attestation Manager. The Attestation Manager component type that is inserted into the model will be the same component type as the communication driver.

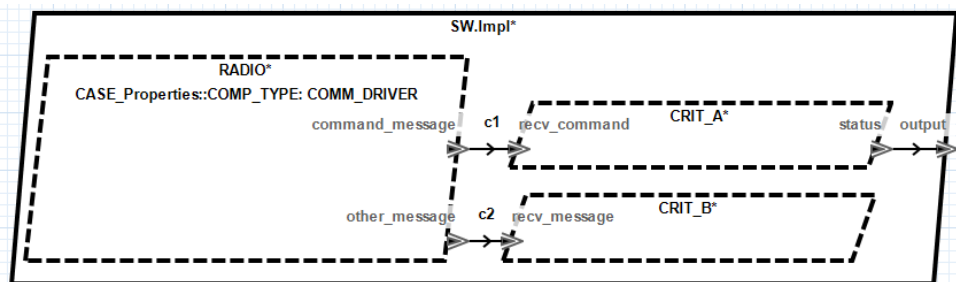
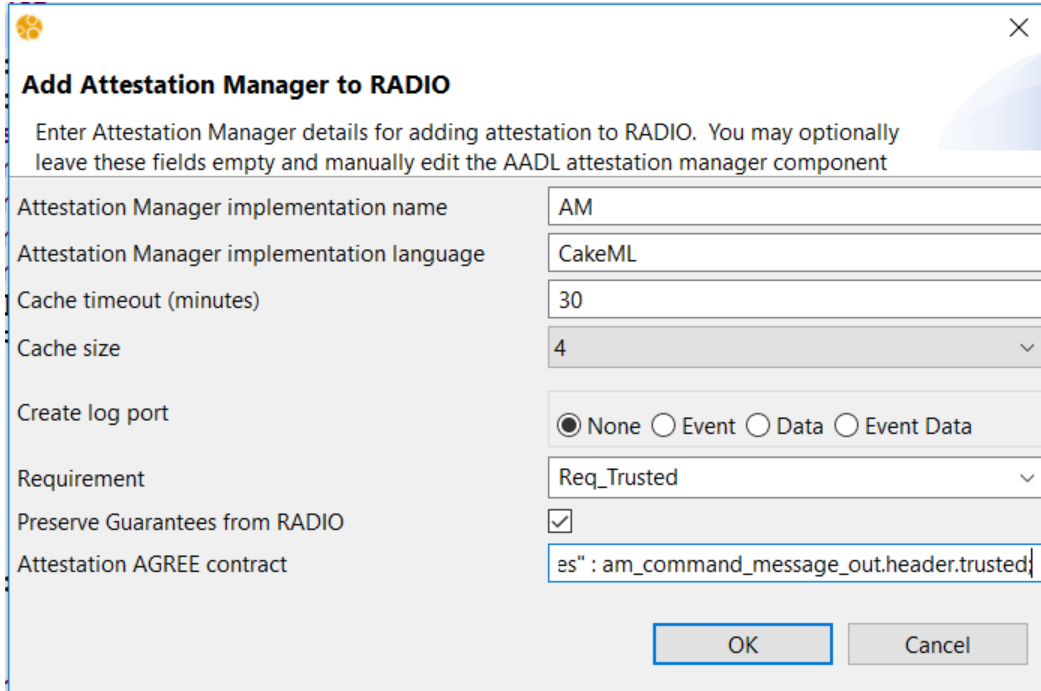


Figure 15. Initial model.

The Attestation Manager is added to the model by selecting a communication driver subcomponent in a component implementation and clicking the CASE → Cyber Resiliency → Model Transformations → Add Attestation Manager... menu item. A wizard will appear, as shown in Figure 16, enabling the user to customize the Attestation Manager.





**Add Attestation Manager to RADIO**

Enter Attestation Manager details for adding attestation to RADIO. You may optionally leave these fields empty and manually edit the AADL attestation manager component

Attestation Manager implementation name	AM
Attestation Manager implementation language	CakeML
Cache timeout (minutes)	30
Cache size	4
Create log port	<input checked="" type="radio"/> None <input type="radio"/> Event <input type="radio"/> Data <input type="radio"/> Event Data
Requirement	Req_Trusted
Preserve Guarantees from RADIO	<input checked="" type="checkbox"/>
Attestation AGREE contract	es\" : am_command_message_out.header.trusted

OK Cancel

Figure 16. Add Attestation Manager wizard.

The Attestation transform will create a special CASE\_AttestationManager AADL component type and implementation, and insert them into the model. It will then instantiate the CASE\_AttestationManager implementation as a subcomponent in the implementation containing the selected communication driver. The user may provide a name for the attestation manager subcomponent, 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.

By default the CASE attestation manager 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 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.

The requirement drop-down box lists all of the cyber-requirements that have been imported from TA1 tools. By specifying the cyber requirement that drives the attestation 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 attestation manager, but it is highly recommended for construction of the proper system assurance case.

Finally, the user may provide an AGREE *guarantee* statement, which is a formal property the attestation manager asserts to always be true as long as stated assumptions on the environment are valid. Specifying such a property is typically done by referring to the outgoing message on the attestation manager’s output port. The message type will be the same as the communication driver component’s output port. Within the AGREE statement, the attestation manager output port name can be referred to by *am\_<Comm\_Driver\_Feature\_Name>\_out*. For the example in Figure 15, if the message header on

connection `c1` has a field to indicate that the message source is trusted, such as on line 16 of the `CASE_Model_Transformations.aadl` package (see Figure 17):

```

9      -- CASE message header
10     data CASE_MsgHeader
11     end CASE_MsgHeader;
12     data implementation CASE_MsgHeader.Impl
13     subcomponents
14         src: data Base_Types::Integer;
15         dst: data Base_Types::Integer;
16         trusted: data Base_Types::Boolean;
17         HMAC: data Base_Types::Boolean;
18     end CASE_MsgHeader.Impl;
19
20     -- CASE RF Message structure
21     data CASE_RF_Msg
22     end CASE_RF_Msg;
23
24     data implementation CASE_RF_Msg.Impl
25     subcomponents
26         header: data CASE_MsgHeader.Impl;
27         payload: data;
28     end CASE_RF_Msg.Impl;

```

Figure 17. RF message definition in `CASE_Model_Transformations.aadl`.

The AGREE statement will then be:

```

guarantee Req_Trusted_Attestation "Only messages from trusted sources shall
be permitted" : am_command_message_out.header.trusted;

```

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

Clicking the OK button on the wizard will insert the `CASE_AttestationManager` and extended communication driver into the model, as shown in Figure 18 and Figure 19, respectively. The graphical representation is shown in Figure 20.

```

9     thread CASE_AttestationManager
10     features
11         am_command_message_in: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
12         am_command_message_out: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
13         am_other_message_in: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
14         am_other_message_out: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
15         am_request: out event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
16         am_response: in event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
17     properties
18         CASE_Properties::COMP_TYPE => ATTESTATION;
19         CASE_Properties::COMP_SPEC => ("Req_Trusted_Attestation");
20     annex agree {
21         guarantee Req_Trusted_Attestation "Command messages shall only come from trusted sources" : am_command_message_out.header.trusted;
22     };
23     end CASE_AttestationManager;
24
25     thread implementation CASE_AttestationManager.Impl
26     properties
27         CASE_Properties::COMP_IMPL => "CafeML";
28         CASE_Properties::CACHE_TIMEOUT => 30;
29         CASE_Properties::CACHE_SIZE => 4;
30     end CASE_AttestationManager.Impl;

```

Figure 18. Line 9: `CASE_AttestationManager` component type; Line 25: `CASE_AttestationManager` component implementation.

```

65Ⓢ  thread RadioDriver_Attestation extends RadioDriver
66      features
67          am_request: in event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
68          am_response: out event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
69      end RadioDriver_Attestation;
70
71Ⓢ  thread implementation RadioDriver_Attestation.Impl extends RadioDriver.Impl
72      end RadioDriver_Attestation.Impl;
73
74Ⓢ  process SW
75      features
76          input: in event data port;
77          output: out event data port Base_Types::Boolean;
78Ⓢ  annex agree {**
79      guarantee "Output status shall never be false" : output;
80      **};
81      end SW;
82
83Ⓢ  process implementation SW.Impl
84      subcomponents
85          RADIO: thread RadioDriver_Attestation.Impl;
86          CRIT_A: thread Critical_A.Impl;
87          CRIT_B: thread Critical_B.Impl;
88          AM: thread CASE_AttestationManager.Impl;
89      connections
90          c1: port input -> RADIO.recv_message;
91          c5: port RADIO.command_message -> AM.am_command_message_in;
92          c6: port RADIO.other_message -> AM.am_other_message_in;
93          c7: port AM.am_request -> RADIO.am_request;
94          c8: port RADIO.am_response -> AM.am_response;
95          c2: port AM.am_command_message_out -> CRIT_A.recv_command;
96          c3: port AM.am_other_message_out -> CRIT_B.recv_message;
97          c4: port CRIT_A.status -> output;
98Ⓢ  annex resolute {**
99      prove Req_Trusted(this.CRIT_A, "Req_Trusted", this.RADIO, this.AM)
100      **};
101      end SW.Impl;

```

Figure 19. Line 65: Extended communication driver component; Line 88: Attestation Manager subcomponent; Lines 91-96: Attestation Manager connections; Line 99: updated assurance claim call.

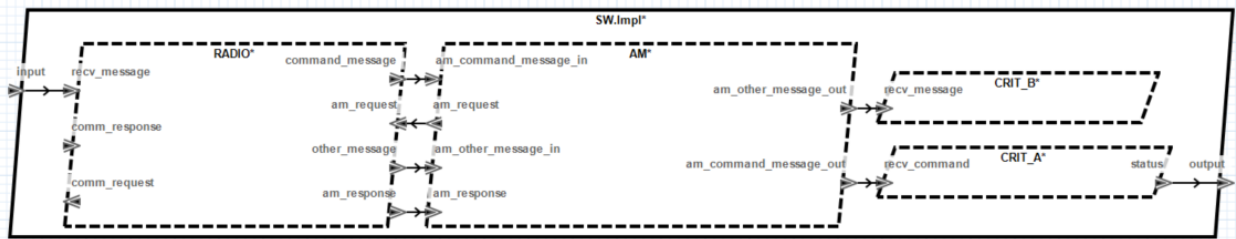


Figure 20. Transformed model containing an attestation manager.

For every outgoing connection from the communication driver to other software components, a corresponding input and output port is created in the attestation manager. The Attestation transform also adds two connections between the Attestation Manager and the communication driver to perform the attestation with the message source. In addition, two corresponding response and request ports are added for communication with the message sender. These ports are intentionally left unconnected. It is up to the user to connect them to a corresponding attestation manager on the system of the message sender, if contained in the model. 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. Note that AADL feature groups are not currently supported for the Attestation transform, but will be in future versions of the tool.

## 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 subclaims as model transformations are performed.

When a requirement is imported from a TA1 tool it will be placed in the CASE\_Requirements package as a Resolute claim, as shown in Figure 21.

```
4  Req_Trusted(comp_context : component, property_id : string) <=
5      ** "[trusted] Command messages shall only come from trusted sources" **
6      context Generated_By : "GearCASE"
7      context Generated_On : "May 23, 1970"
8      context Req_Component : "Attestation::SW.Impl.CRIT_A"
9      context Formalized : "True"
10     agree_prop_checked(comp_context, property_id)
```

Figure 21. 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 Attestation transform is applied, the requirement is updated with an additional check to make, which reflects the addition of the attestation manager component, as shown in Figure 22.

```
4  Req_Trusted(comp_context : component, property_id : string, comm_driver : component, attestation_manager : component)
5      ** "[trusted] Command messages shall only come from trusted sources" **
6      context Generated_By : "GearCASE"
7      context Generated_On : "May 23, 1970"
8      context Req_Component : "Attestation::SW.Impl.CRIT_A"
9      context Formalized : "True"
10     agree_prop_checked(comp_context, property_id) and add_attestation_manager(comm_driver, attestation_manager)
```

Figure 22. Modified requirement after Attestation transform.

The addition of the `add_attestation_manager()` call on line 10 provides Resolute with additional checks to make to ensure the requirement was addressed correctly. In this case, `add_attestation_manager()` is included in the CASE\_Model\_Transformations library and consists of two subclaims:

- `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

To check whether the requirement has been correctly addressed in the design, select the containing component implementation (SW.Impl on line 83 in Attestation.aadl) and select Analyses → Resolute from the main menubar. The Resolute output will appear in the output pane, as shown in Figure 23.

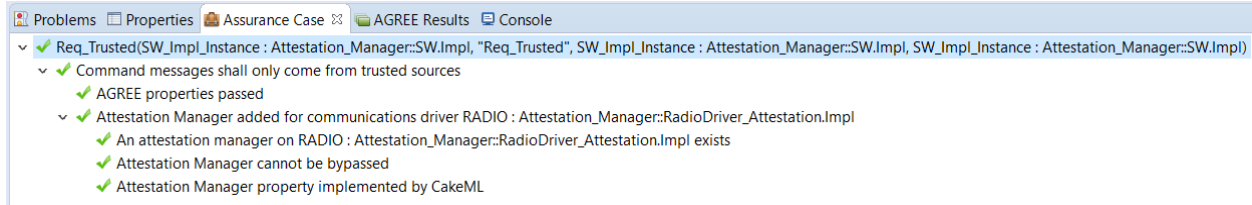


Figure 23. A passing Resolute analysis.

## 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_Transformations/Virtualization/Simple_Example)

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.

The Virtualization transform can also be performed on the CASE Phase 1 UAV example model. Three versions of the model are available for reference:

- Initial model – This is the Phase 1 UAV model that includes an imported cyber requirement, which drives the binding of the FlightPlanner component to a virtual processor. The Initial model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Virtualization/UAV\\_Example/Initial\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Virtualization/UAV_Example/Initial_Model)
- Transformed model – This is the Phase 1 UAV model after the Isolator transform has been applied. The Transformed model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Virtualization/UAV\\_Example/Transformed\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Virtualization/UAV_Example/Transformed_Model)
- Test model – This is the Phase 1 UAV model containing several implementations for testing the correctness of the Resolute evaluation on the Virtualization transform. The Test model can be found here:  
[https://github.com/loonwerks/CASE/tree/master/TA2/Model\\_Transformations/Virtualization/UAV\\_Example/Test\\_Model](https://github.com/loonwerks/CASE/tree/master/TA2/Model_Transformations/Virtualization/UAV_Example/Test_Model)

Processes, threads, and thread groups can be bound to a virtual machine. Note that to isolate 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 69 of Virtualization.aadl (see Figure 24).

```

64  system implementation Critical.Impl
65      subcomponents
66          PROC : processor HW_Proc.Impl;
67          SW : process SW.Impl;
68      properties
69          Actual_Processor_Binding => (reference (PROC)) applies to SW;
70  annex resolute {**
71      prove(Req_Virtualization(this.SW))
72      **};
73  end Critical.Impl;

```

Figure 24. Line 69: Software process is bound to hardware processor component via *Actual\_Processor\_Binding* property association.

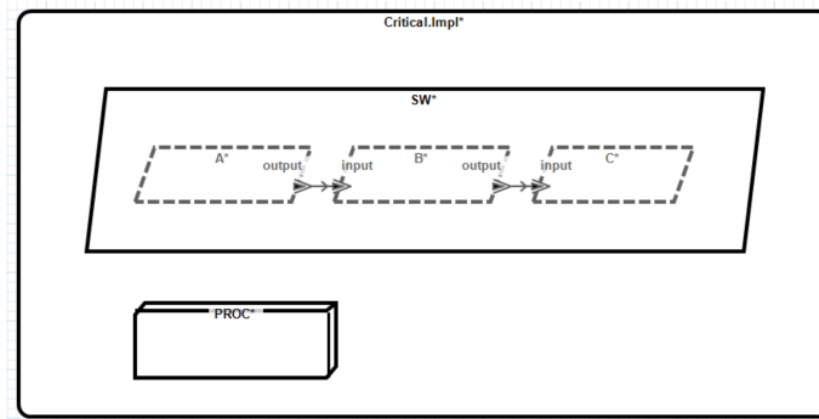
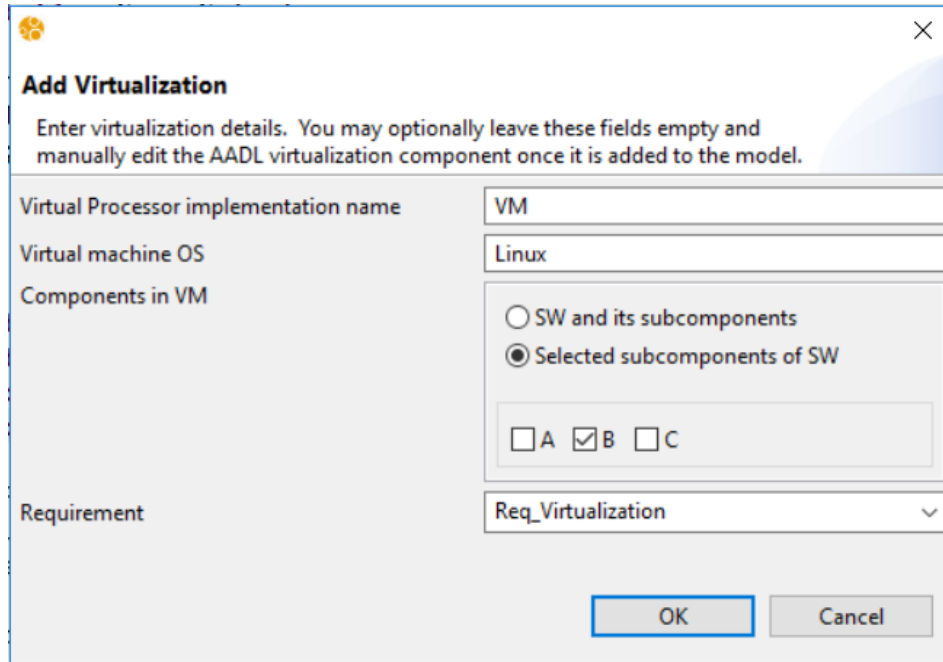


Figure 25. Initial model.

To apply the Virtualization transform, select a process, thread, or thread group subcomponent in a component implementation (for example, in *Virtualization.aadl*, select the SW subcomponent on line 67). 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 CASE → Cyber Resiliency → Model Transformations → Add Virtualization... menu item. A wizard will appear, as shown in Figure 26.



**Add Virtualization**

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

Virtual Processor implementation name	VM
Virtual machine OS	Linux
Components in VM	<input type="radio"/> SW and its subcomponents <input checked="" type="radio"/> Selected subcomponents of SW <div style="border: 1px solid gray; padding: 5px; margin-top: 5px;"> <input type="checkbox"/> A <input checked="" type="checkbox"/> B <input type="checkbox"/> C         </div>
Requirement	Req_Virtualization

Figure 26. 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 subcomponent, or use the default name in the first field of the wizard. 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 build process will package bound components in a virtual machine. You can specify the virtual machine operating system, use the default name, or leave blank. 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 of the imported cyber-requirements from the TA1 tools. 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 CASE\_Virtual\_Machine component type and component implementation are added to the AADL file, and a CASE\_Virtual\_Machine subcomponent is inserted into the component implementation containing the software component that was selected for virtualization (see Figure 27). The graphical representation is shown in Figure 28.



```

62 virtual processor CASE_Virtual_Machine
63   properties
64     CASE_Properties::COMP_TYPE => VIRTUAL_MACHINE;
65 end CASE_Virtual_Machine;
66
67 virtual processor implementation CASE_Virtual_Machine.Impl
68   properties
69     CASE_Properties::OS => "Linux";
70 end CASE_Virtual_Machine.Impl;
71
72 system Critical
73 end Critical;
74
75 system implementation Critical.Impl
76   subcomponents
77     PROC: processor HW_Proc.Impl;
78     SW: process SW.Impl;
79     VM: virtual processor CASE_Virtual_Machine.Impl;
80   properties
81     Actual_Processor_Binding => (reference (PROC)) applies to SW, VM;
82     Actual_Processor_Binding => (reference (VM)) applies to SW.B;
83 annex resolute {**
84   prove Req_Virtualization(this.SW, {this.SW.B}, this.VM)
85 **};
86 end Critical.Impl;

```

Figure 27. Line 62: CASE\_Virtual\_Machine component type; Line 67: CASE\_Virtual\_Machine component implementation; Line 79: CASE\_Virtual\_Machine subcomponent; Lines 81-82: updated processor bindings.

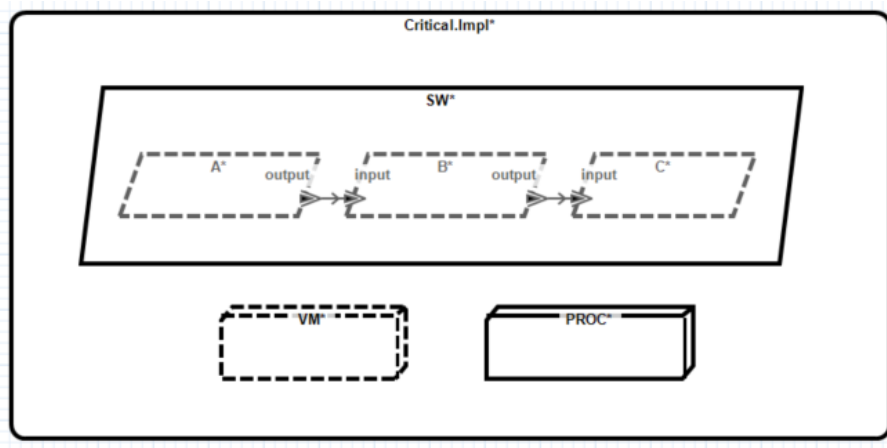


Figure 28. Transformed model containing a virtual machine.

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 implicitly applied to 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` subclaim from the `CASE_Model_Transformations` claim library (see Figure 29). 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.

```
4  Req_Virtualization(comp_context : component, bound_components : {component}, virtual_machine : component) <=
5      ** "[virtualization] Third-party software shall be isolated from critical components" **
6      context Generated_By : "GearCASE"
7      context Generated_On : "May 23, 1970"
8      context Req_Component : "Virtualization::Critical.Impl.SW"
9      context Formalized : "False"
10     add_virtualization(bound_components, virtual_machine)
```

Figure 29. Virtualization requirement in Resolute.

## 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, runtime preferences need to be set. To do so, select **Window** → **Preferences** from the main menubar, expand the **CASE** item on the left-hand side of the window, and select **SPLAT** (see Figure 30).

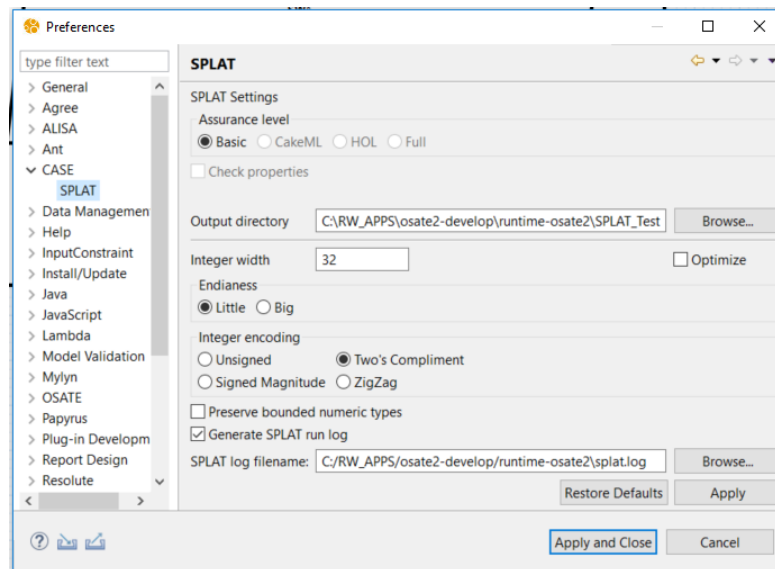
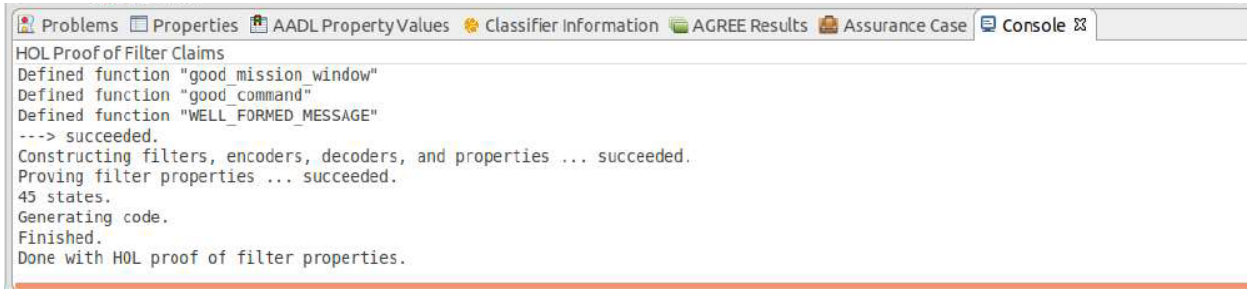


Figure 30. SPLAT preferences.

- *Assurance level* describes how SPLAT will generate the filter implementation:
  - **Basic:** PolyML regexp compiler (which was generated from HOL model) builds DFA, from which {C, Ada, Java, SML} can be generated. Currently, this option only produces C source.
  - **CakeML:** Input regexp constructed by splat frontend; then passed via hol2deep sexp interface. Then off-the-shelf cake binary loaded with compiled regexp compiler applied to regexp to get DFA and then binary. Produces .o file for linking with other C code.
  - **HOL:** HOL regexp compiler run inside logic to get DFA. The DFA is then pushed through hol2deep, generating DFA.sexp, then CakeML binary invoked, as for the cake option.
  - **Full:** Similar to the *HOL* option, except that the CakeML compiler is applied inside-the-logic so that correctness of regexp compilation can be joined to the ML compiler correctness theorem to get an end-to-end correctness theorem. Thus no invocation of hol2deep or generation of .sexp version or invocation of off-the-shelf compilers. Produces .o file for linking with other C code and proofs.
- Selecting the *Check properties* checkbox will cause SPLAT to conduct automated proofs in order to check that the generated regexp indeed implements the well-formedness specifications that it was generated from. Properties are not checked by default.
- 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 input specifications need not constrain the *Integer width*, so this can be specified (in bits). If option *optimize* is specified, then the width of each field will be the least number of bytes needed to express the numbers allowed by the AGREE specification of the field. Otherwise, each integer field is of the specified width. The default value is 32, with no optimization.
- Multi-byte integer representations need *Endianness* to be specified. The default is little endian.
- The representation for *Integer encoding* can be specified to a fixed number of choices. The default encoding is two's complement.

- The kinds of number allowed in declarations of an AADL model typically allow the user to specify different widths and signs, e.g., Integer\_32 and Unsigned\_64. In contrast, AGREE only operates over the unbounded Integer type, which places no restrictions on the size of numbers. This creates a potential mismatch between the objects being reasoned about and the AGREE specification language. By default, numeric types and expressions are promoted to Integer; this is convenient for SPLAT, and in line with the behavior of AGREE, but has the potential to lose valuable information. To override this behavior, select *Preserve bounded numeric types*.
- 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 menubar. 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 31).



```
Problems Properties AADL PropertyValues 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 31. 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

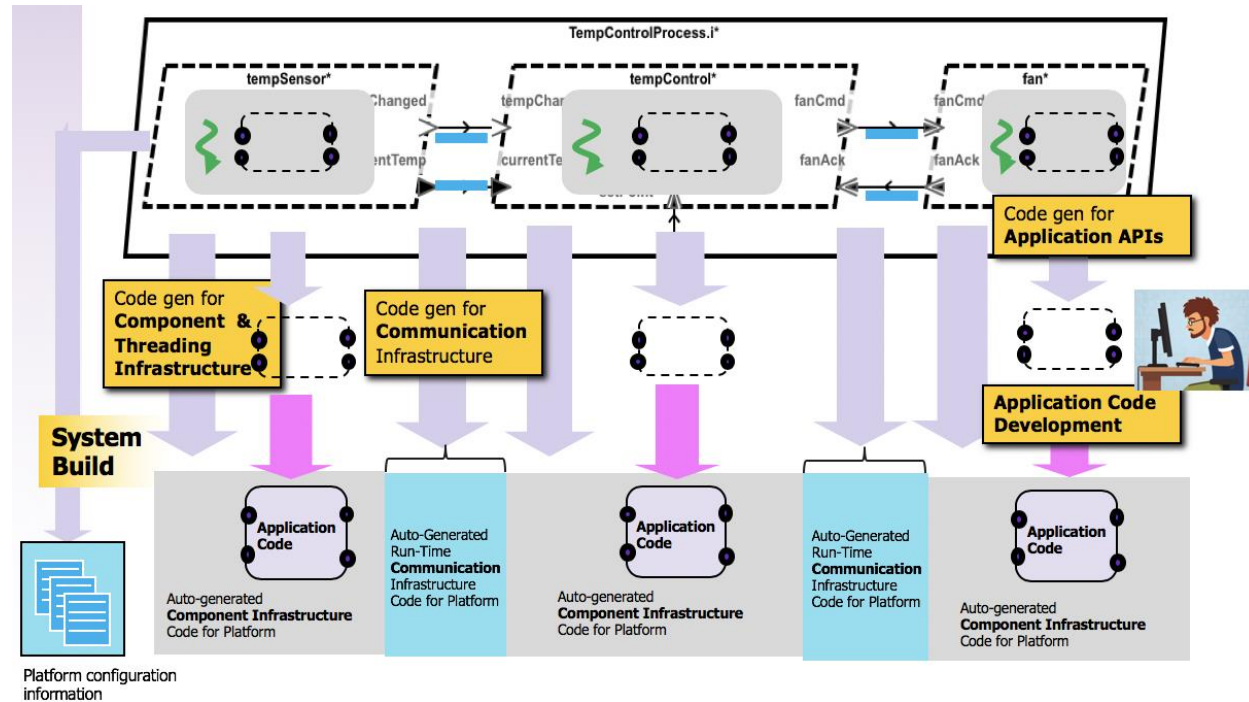


Figure 32. HAMR Translation Architecture.

Figure 32 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 32 – 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. StairCASE-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](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](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.

StairCASE 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.