

BriefCASE Model Transformations

June 2021

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



Contents

| | |
|--------------------------------|----|
| Introduction | 2 |
| 1. Model Transformations | 2 |
| Filter | 2 |
| CASE Filter Properties | 5 |
| Filter Synthesis | 6 |
| Design Assurance | 6 |
| Attestation | 7 |
| Design Assurance | 11 |
| Virtualization | 12 |
| Design Assurance | 16 |
| Monitor | 16 |
| Design Assurance | 23 |
| seL4 | 25 |
| Design Assurance | 29 |

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.

This document describes the BriefCASE model transformations.

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 1 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.

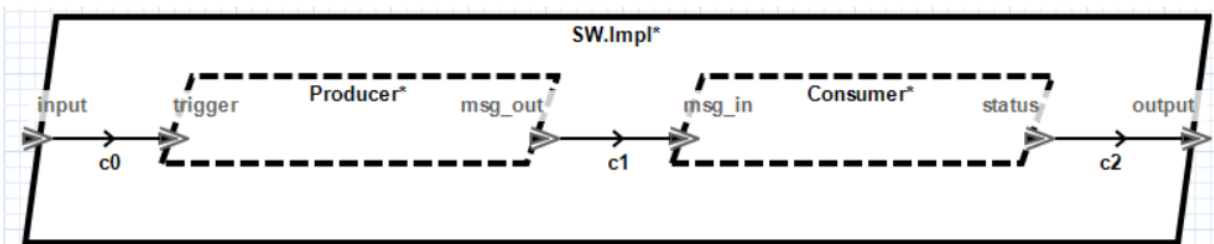


Figure 1. Initial model.

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 2. The wizard enables the user to customize the filter, including providing the filter specification.

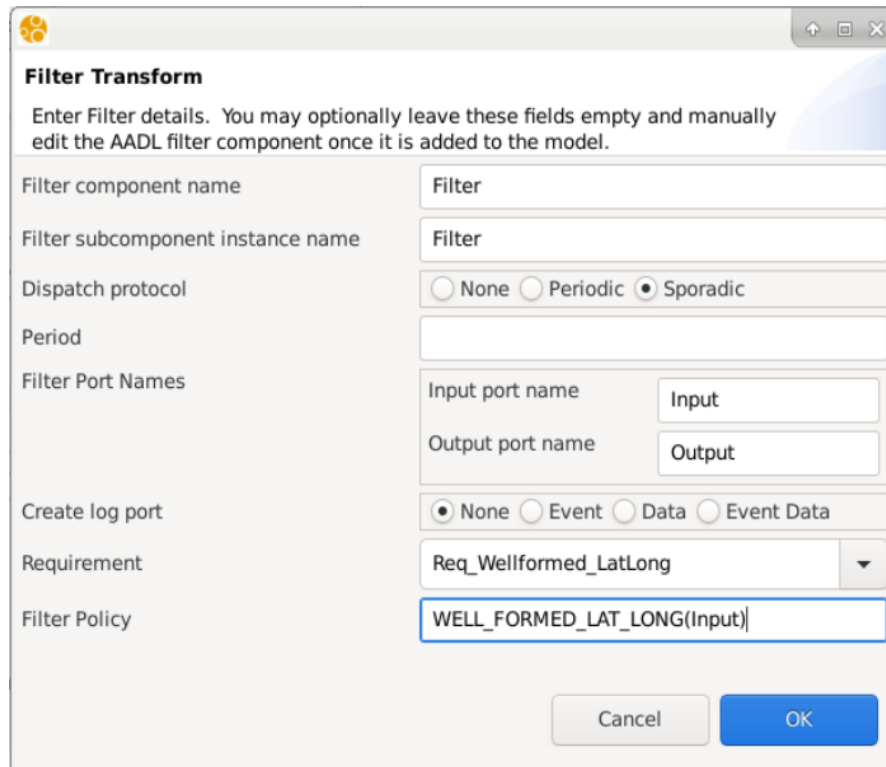


Figure 2. Add Filter wizard

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 3 and Figure 4. The graphical representation is shown in Figure 5.

```

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

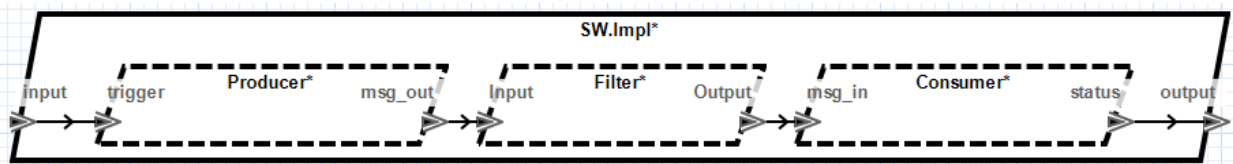


Figure 5. Transformed model containing a filter.

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

3, 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 6.

```

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 6. 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 7.

```

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 7. 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 8.

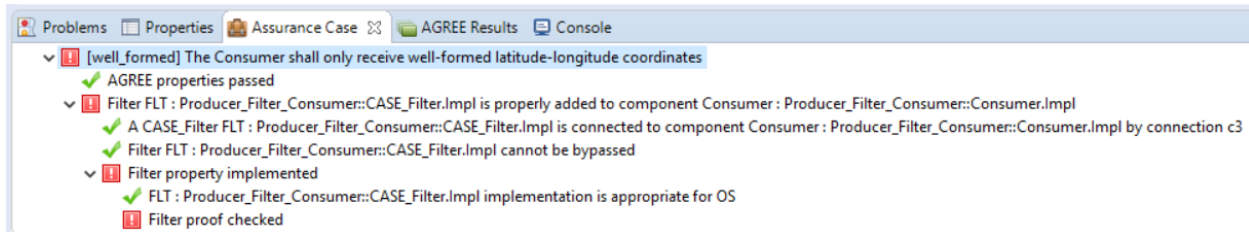


Figure 8. A failing Resolute analysis.

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 9.

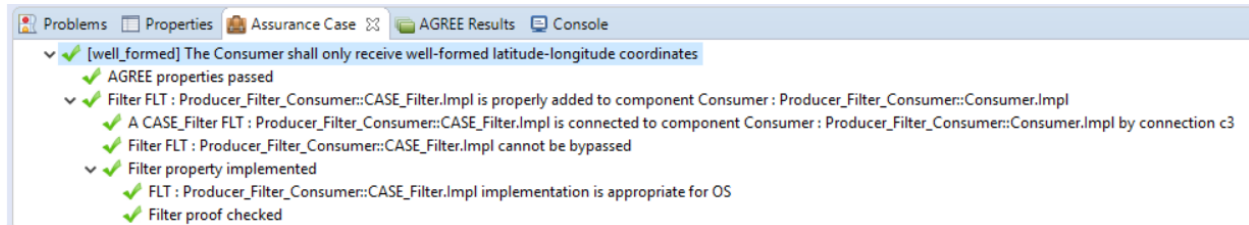


Figure 9. A passing Resolute analysis.

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.

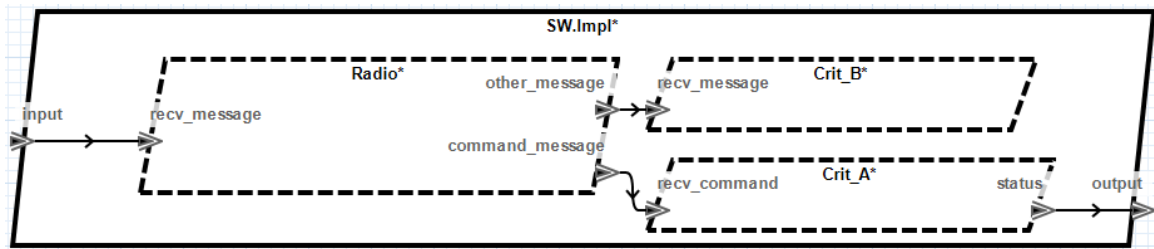
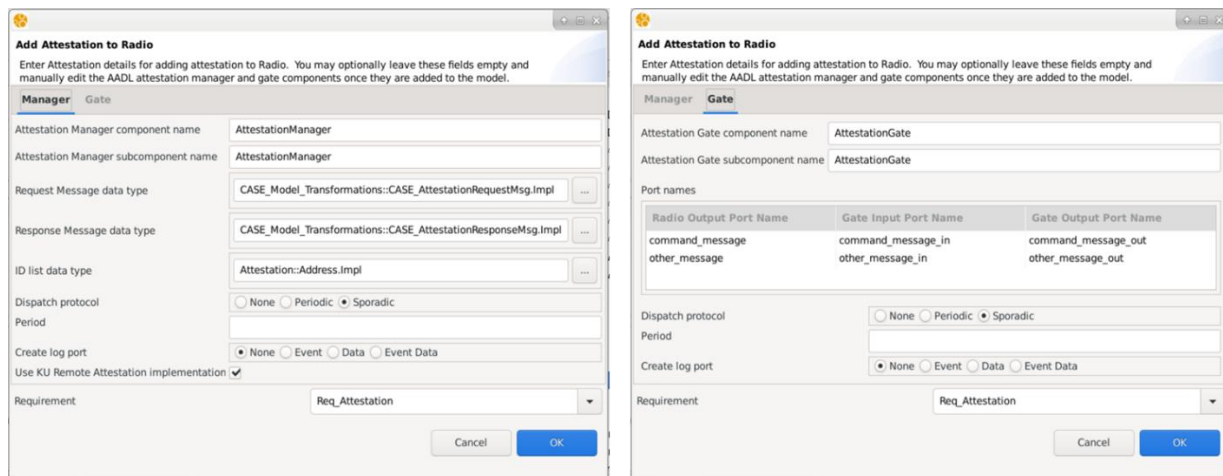


Figure 10. Initial model.

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 11, 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.



Add Attestation to Radio

Enter Attestation details for adding attestation to Radio. You may optionally leave these fields empty and manually edit the AADL attestation manager and gate components once they are added to the model.

Manager | Gate

Attestation Manager component name:

Attestation Manager subcomponent name:

Request Message data type: ...

Response Message data type: ...

ID list data type: ...

Dispatch protocol: ☐ None ☐ Periodic ☒ Sporadic

Period:

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

Use KU Remote Attestation implementation: ☒

Requirement:

Add Attestation to Radio

Enter Attestation details for adding attestation to Radio. You may optionally leave these fields empty and manually edit the AADL attestation manager and gate components once they are added to the model.

Manager | **Gate**

Attestation Gate component name:

Attestation Gate subcomponent name:

| Radio Output Port Name | Gate Input Port Name | Gate Output Port Name |
|------------------------|----------------------|-----------------------|
| command_message | command_message_in | command_message_out |
| other_message | other_message_in | other_message_out |

Dispatch protocol: ☐ None ☐ Periodic ☒ Sporadic

Period:

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

Requirement:

Figure 11. 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 12. Figure 13 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 14). 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 15.

```

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 12. 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 13. 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.

```

57 thread RadioDriver_Attestation
58   features
59     rcv_message: in event data port;
60     command_message: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
61     other_message: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
62     AttestationRequest: in event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
63     AttestationResponse: out event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
64   properties
65     CASE_Properties::Comm_Driver => true;
66 end RadioDriver_Attestation;

```

Figure 14. Line 57: New comm driver definition with attestation ports (lines 62,63).

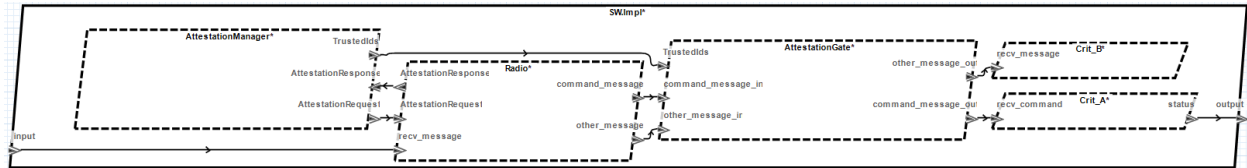


Figure 15. Transformed model containing an attestation manager.

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 16.

```

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 16. 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 17.

```

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 17. 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 18.

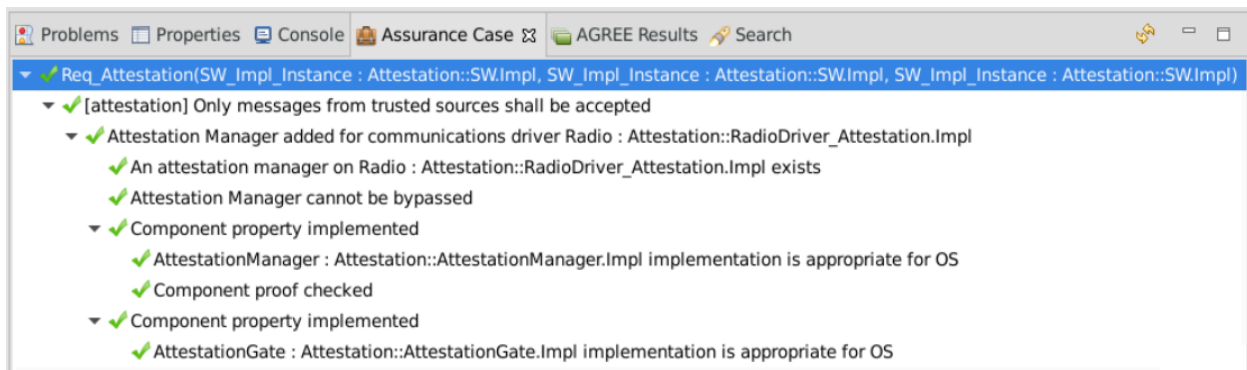


Figure 18. A passing Resolute analysis.

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 19).

```

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

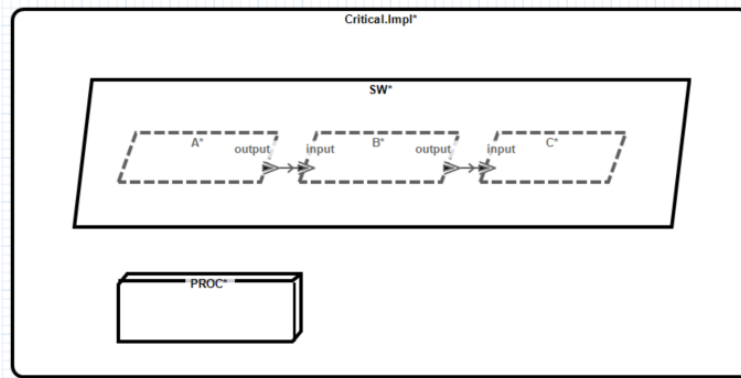


Figure 20. Initial model.

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 21.

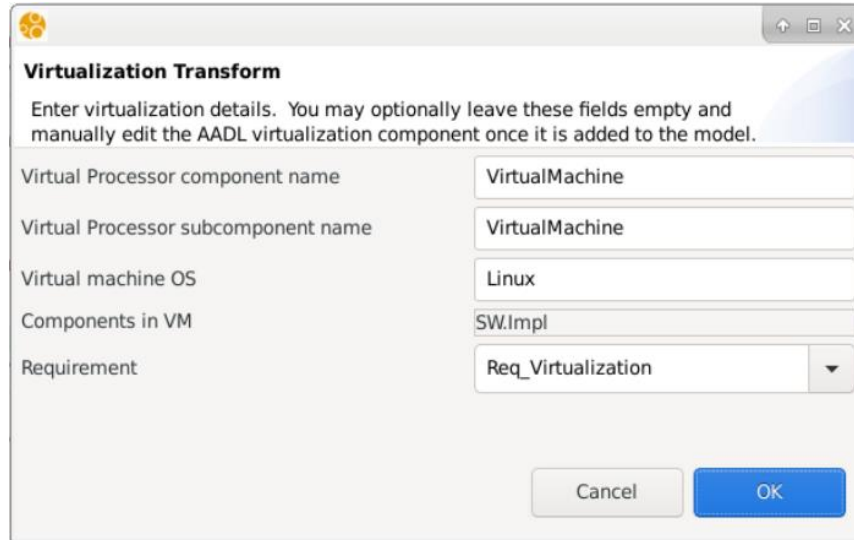


Figure 21. Add Virtualization wizard.

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 22). The graphical representation is shown in Figure 23.


```

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

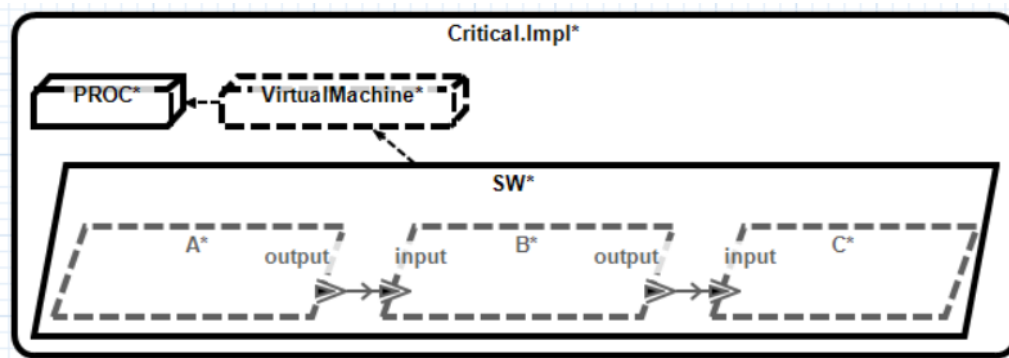


Figure 23. Transformed model containing a virtual machine.

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 24). 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 24. 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 25.

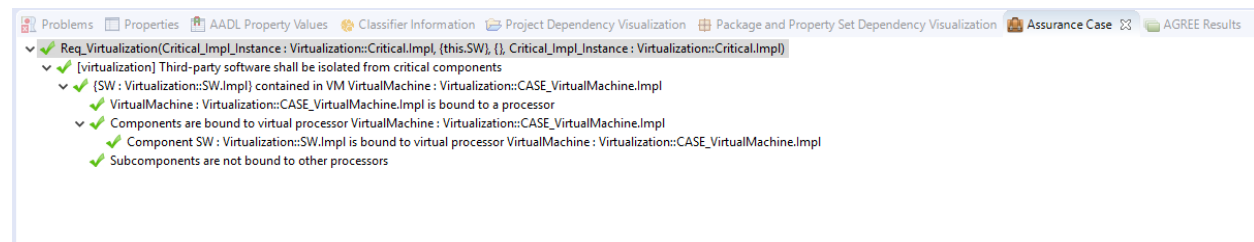


Figure 25. A passing Resolute analysis.

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).

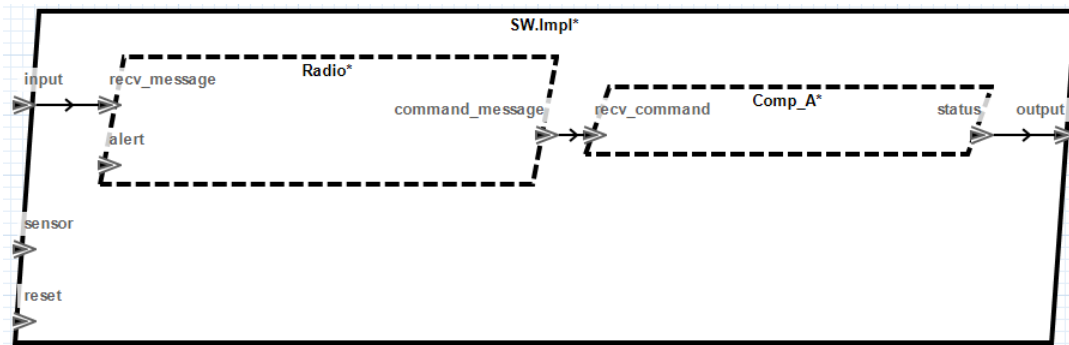


Figure 26. Initial model.

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 27. The wizard enables the user to customize the monitor, including providing the monitor policy specification.

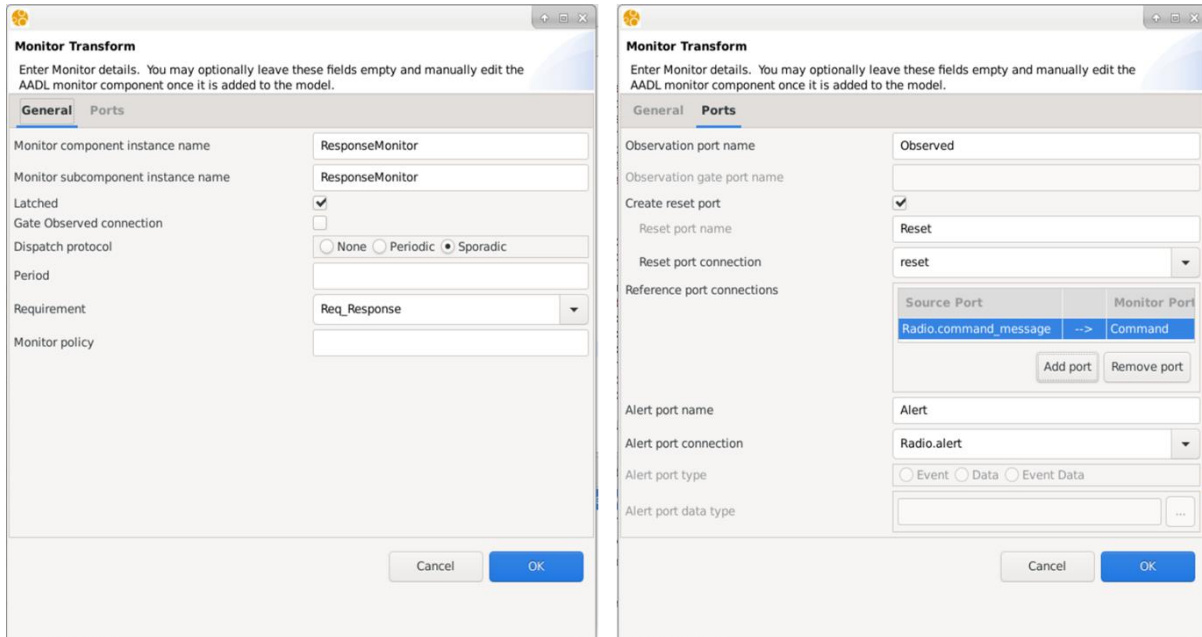


Figure 27. Add Monitor wizard for Response Monitor.

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 28 and Figure 29. The graphical representation is shown in Figure 30.

```

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

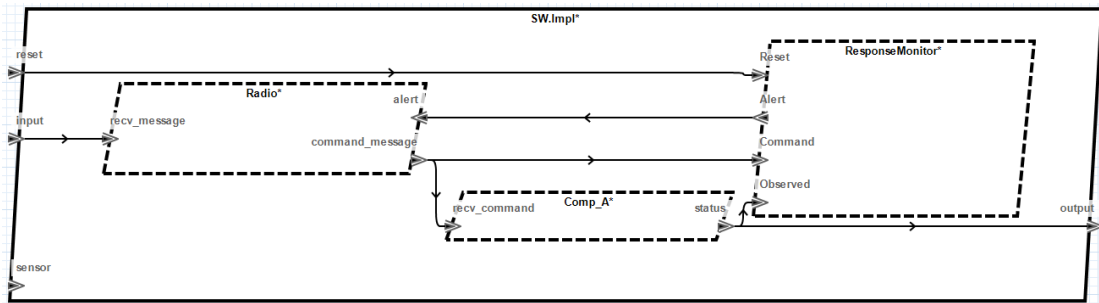


Figure 30. Transformed model containing a response monitor.

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 31.

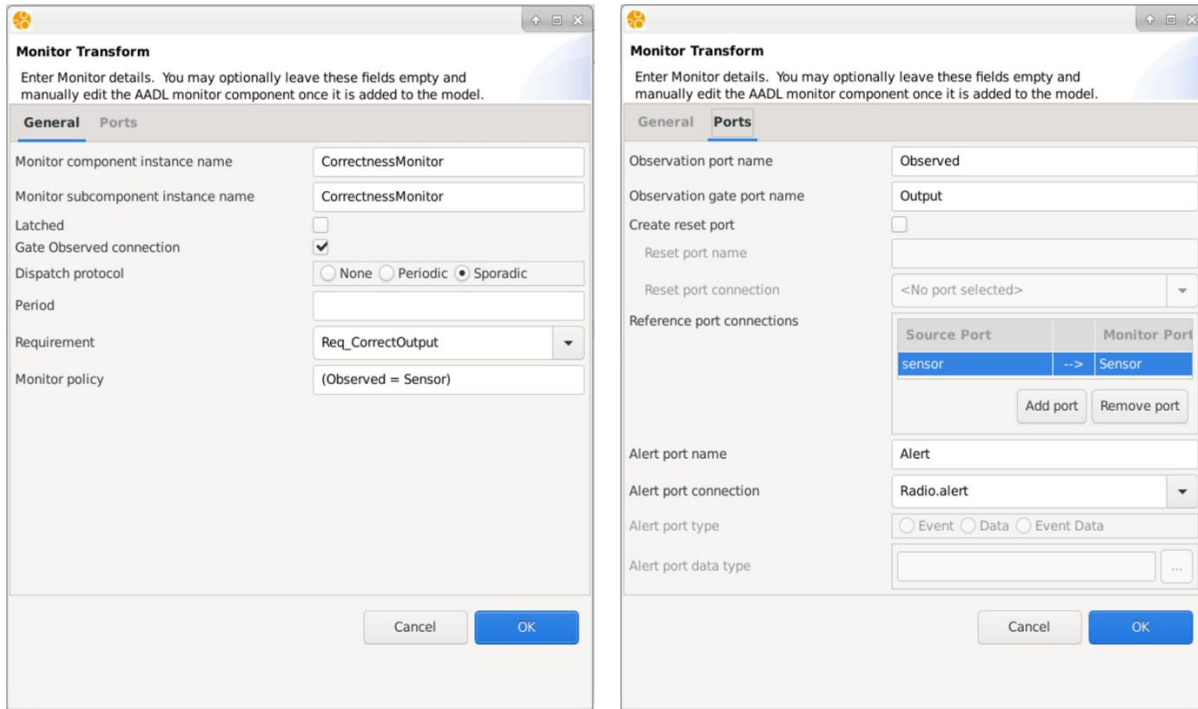
```

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 31. 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 32.



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

General | Ports

Monitor component instance name: CorrectnessMonitor

Monitor subcomponent instance name: CorrectnessMonitor

Latched: ☐

Gate Observed connection: ☒

Dispatch protocol: ☐ None ☐ Periodic ☒ Sporadic

Period:

Requirement: Req_CorrectOutput

Monitor policy: (Observed = Sensor)

Cancel OK

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

General | **Ports**

Observation port name: Observed

Observation gate port name: Output

Create reset port: ☐

Reset port name:

Reset port connection: <No port selected>

Reference port connections:

| Source Port | Monitor Port |
|-------------|--------------|
| sensor | --> Sensor |

Add port Remove port

Alert port name: Alert

Alert port connection: Radio.alert

Alert port type: ☐ Event ☐ Data ☐ Event Data

Alert port data type:

Cancel OK

Figure 32. Wizard for adding a gated monitor.

Clicking the OK button on the wizard will insert the Monitor into the model, as shown in Figure 33 and Figure 34. The graphical representation is shown in Figure 35.

```

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

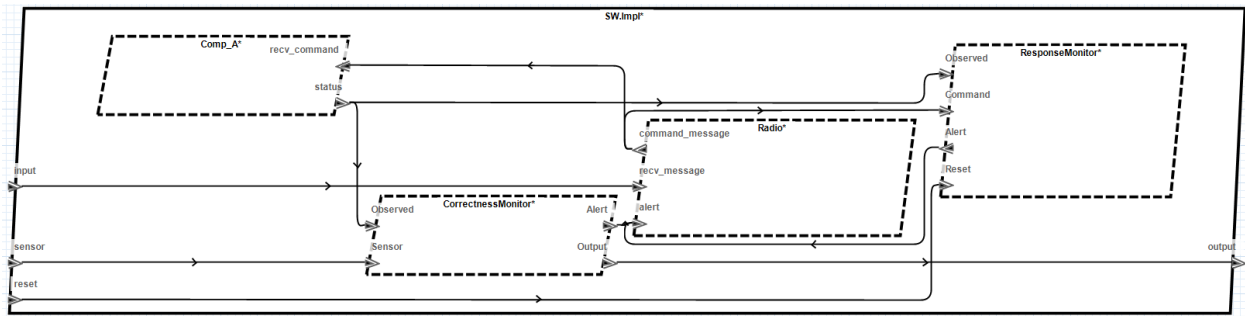


Figure 35. Transformed model containing a correctness monitor.

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 36.


```

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 36. 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 37.

```

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 37. 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 38.

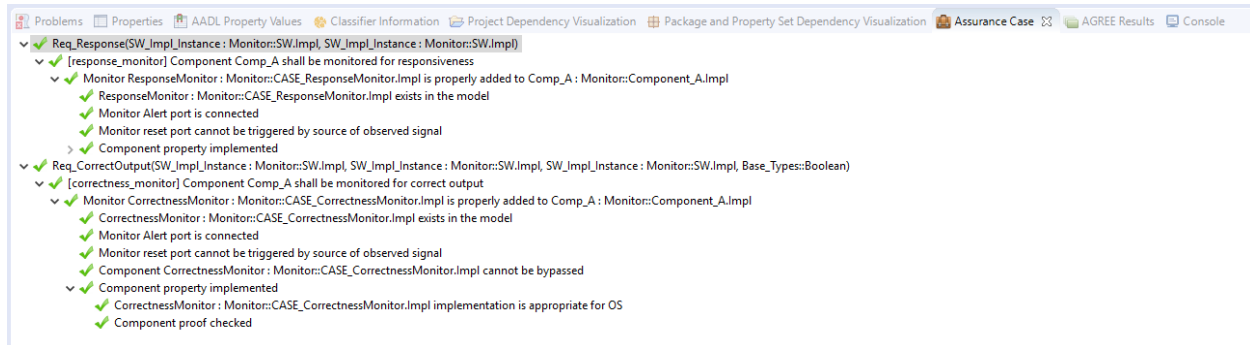


Figure 38. A passing Resolute analysis.

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 39 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.

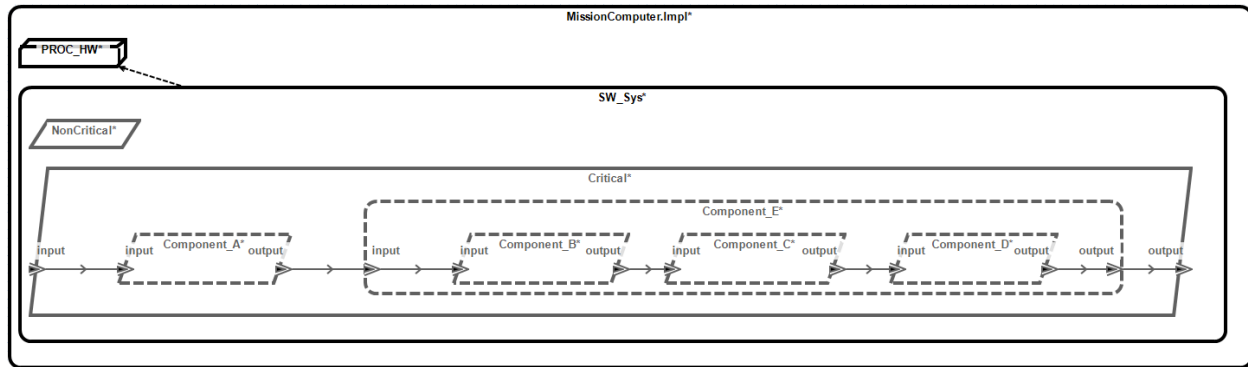


Figure 39. Initial model.

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)

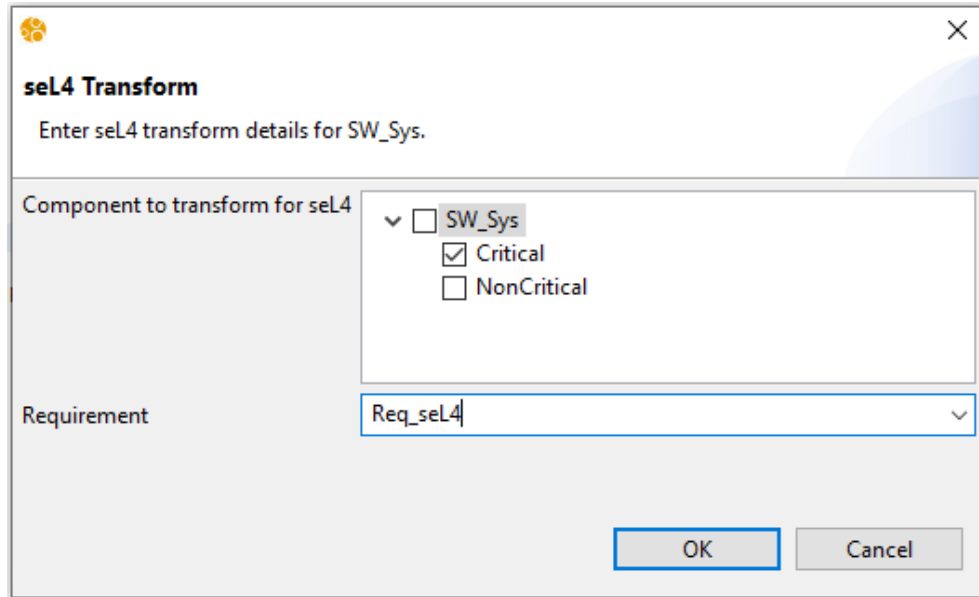
The seL4 transform will result in a subsystem in which:

- 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 40.



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 on the left. The tree shows "SW_Sys" with a checkbox next to it, and under it, "Critical" (checked) and "NonCritical" (unchecked). The bottom section is labeled "Requirement" and contains a text box with "Req_seL4" and a dropdown arrow on the right. At the bottom right of the dialog are two buttons: "OK" and "Cancel".

Figure 40. 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 41. The graphical representation is shown in Figure 42.

```

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 41. 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.

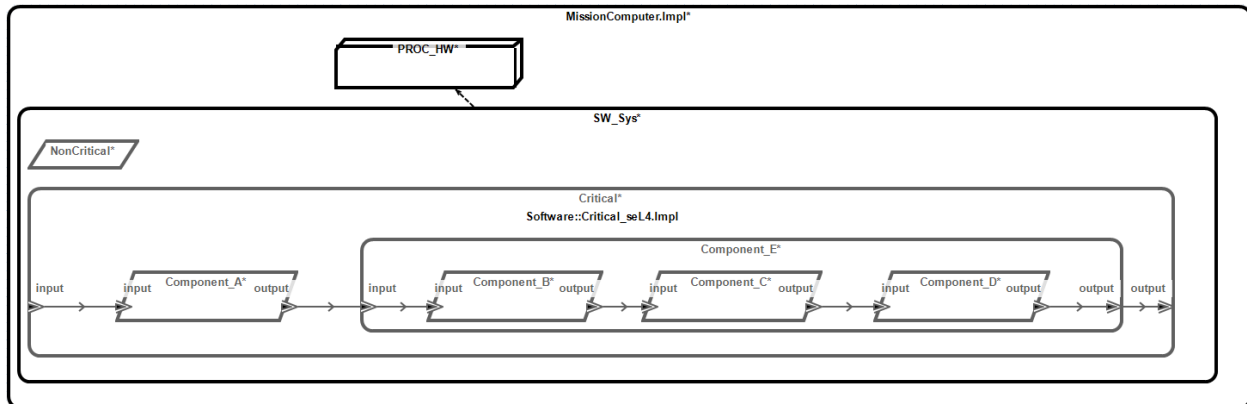


Figure 42. Transformed model.

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 43.

```

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 43. 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 44.

```

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 44. 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 45.

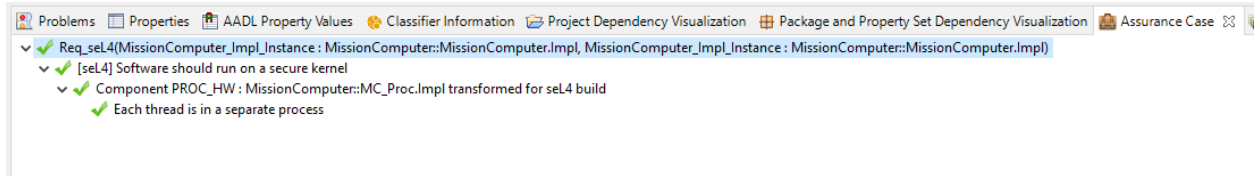


Figure 45. A passing Resolute analysis.