Collins CASE TA2 Filter Transform

|  |
| --- |
| To illustrate the Filter transform, we use a simple producer-consumer example model, which 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.   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/%20UAV_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/%20UAV_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> |

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.

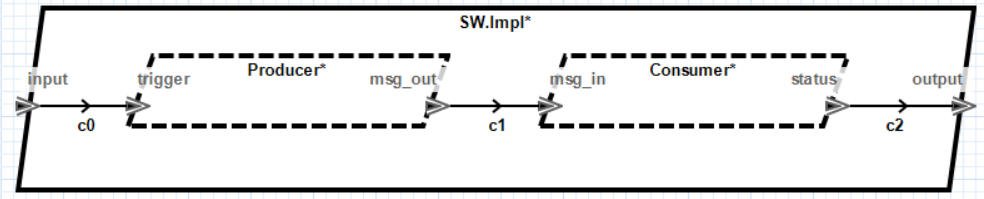


Figure 1. Initial model.

A filter can be added to the following AADL components:

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

The model transform will insert a filter component that has the same type as the 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. 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 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 2. The wizard enables the user to customize the filter, including providing the filter specification.

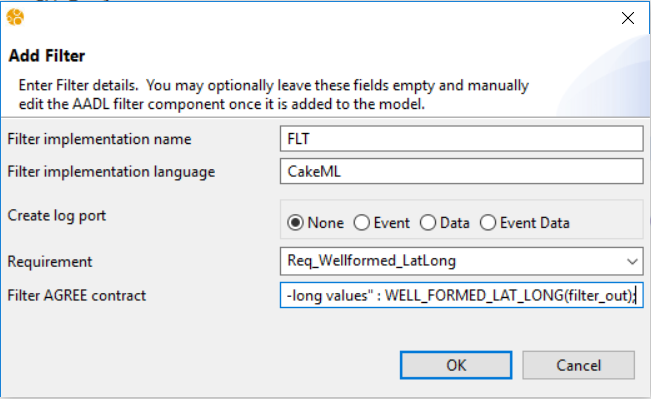


Figure 2. Add Filter wizard

The Filter transform will create 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 3 and Figure 4.

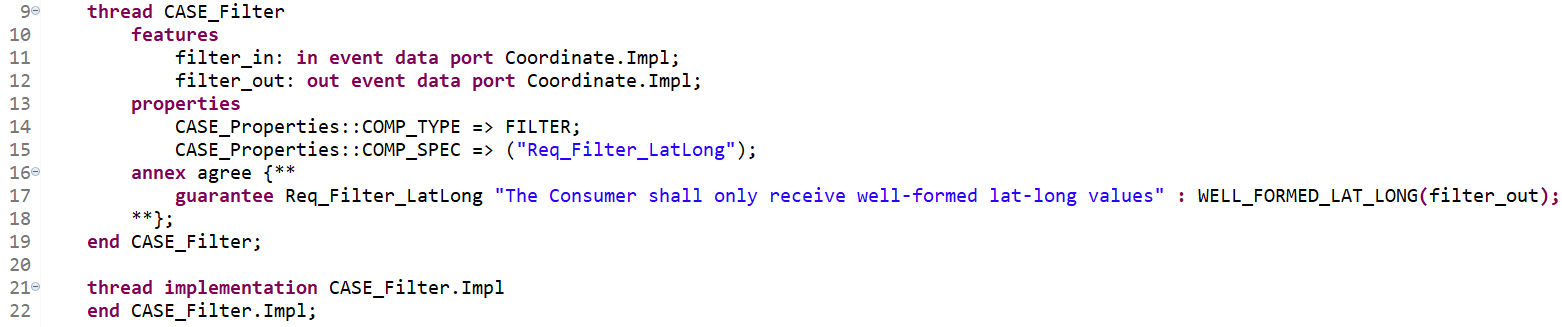


Figure 3. Line 9: CASE\_Filter component type; Line 21: CASE\_Filter component implementation.

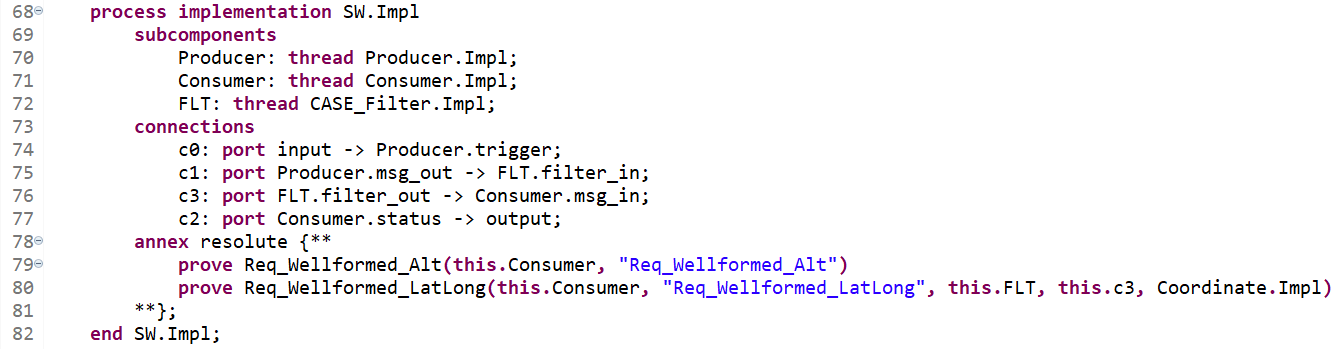


Figure 4. Line 72: filter subcomponent; Lines 75-76: filter connections; Line 80: updated assurance claim call.

### 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 5. A compound filter is simply a single filter component containing multiple filter expressions.

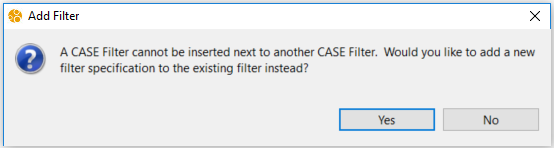


Figure 5. Adjacent CASE Filter warning.

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 6). 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.

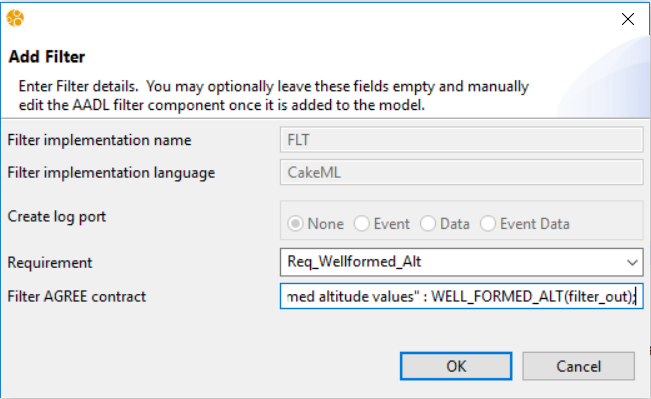


Figure 6. Adding an expression to an existing filter.

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 7.

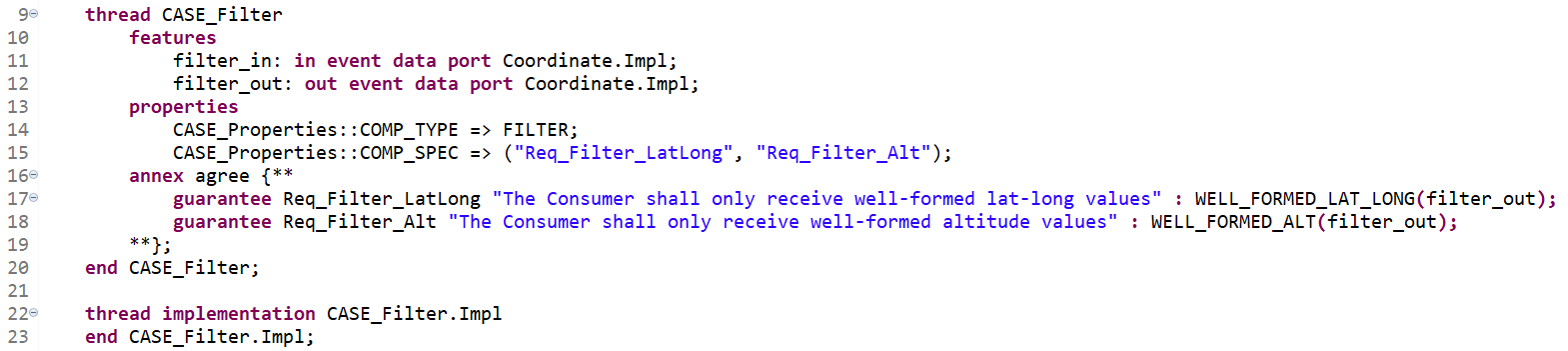


Figure 7. Compound filter.

### 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 7, 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. To run SPLAT, select CASE 🡪 Cyber Resiliency 🡪 Proofs 🡪 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 8).

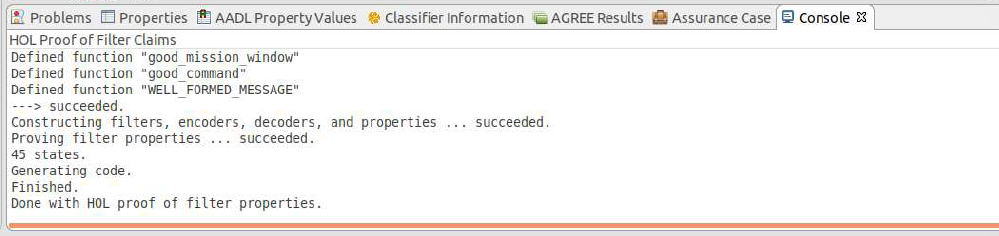


Figure 8. SPLAT status.

If SPLAT runs successfully, the directory containing the TA2 tool executable 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

A test harness can be placed around the 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.

### 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 9.

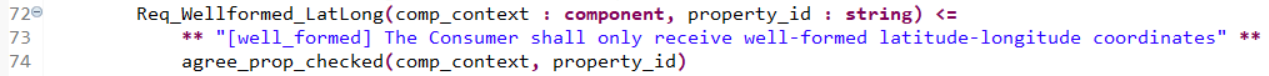


Figure 9. Requirement imported from a TA1 tool.

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 10.

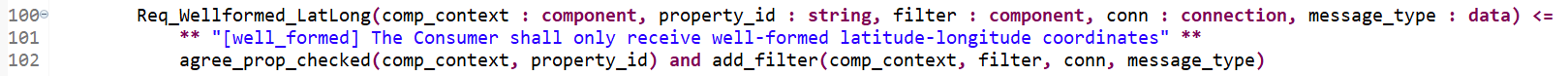


Figure 10. Modified requirement after Filter transform.

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 11.

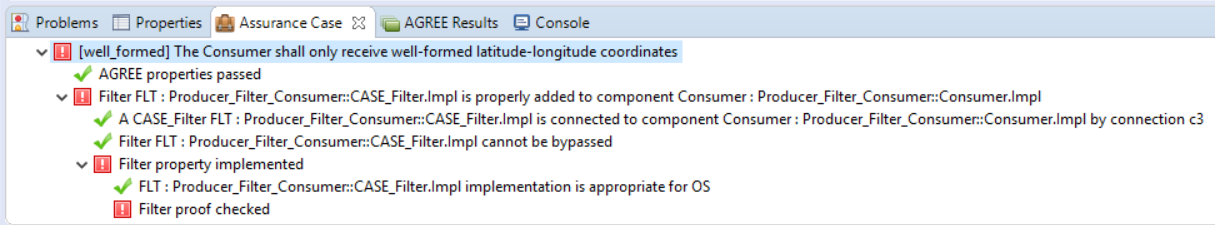


Figure 11. A failing Resolute analysis.

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 12.

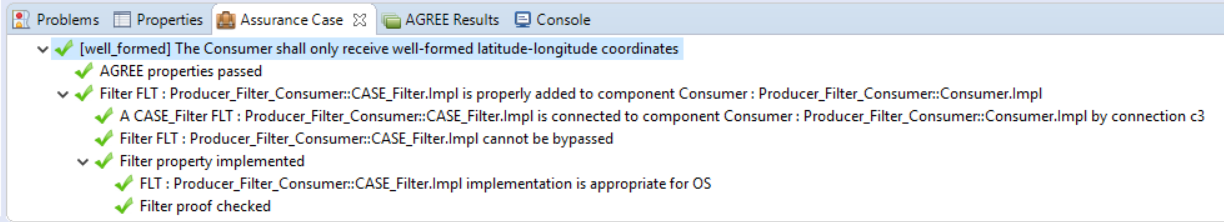


Figure 12. A passing Resolute analysis.