

CASE Tool Demo

Rockwell Collins – TA2

Introduction

In order to gain alignment around tool interfaces and interoperability, we have developed a simple example architecture of a UAV surveillance system. In our scenario, a UAV is used to conduct surveillance over a specified region. A ground station transmits a surveillance region and flight pattern to a UAV, from which a flight mission is generated and executed by the flight controller. The surveillance region may be annotated with no-fly zones and other features relevant to the mission. The flight pattern is a description of the intended UAV behavior, such as *zig-zag across surveillance region*, or *follow perimeter*. The Flight Planner on board the UAV Mission Computer takes the surveillance region and flight pattern input and generates the flight mission, which is a list of waypoints the UAV must follow. The Waypoint Manager passes the current window of waypoints to the UAV Flight Controller.

A high-level view of the system is illustrated in Figure 1.

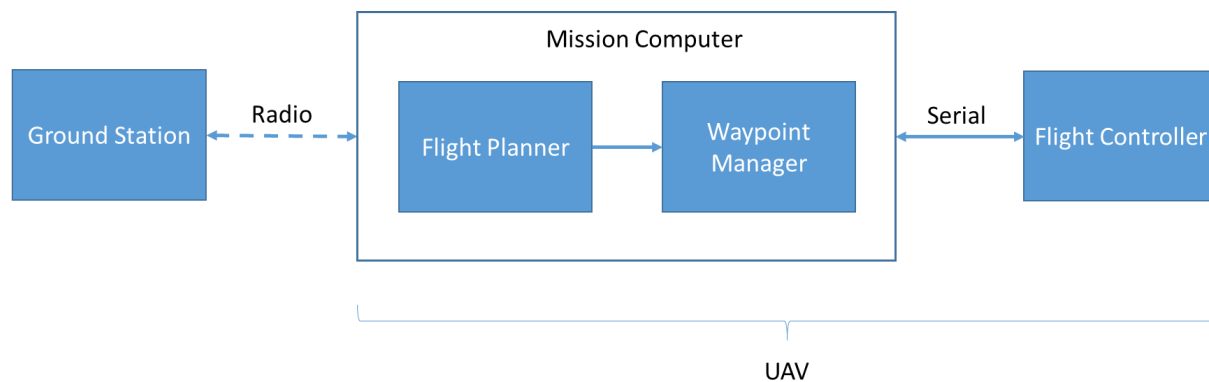


Figure 1. Waypoint Navigation System

Using CASE Tools to Develop a Waypoint Navigation System

To illustrate how we envision incorporating output from TAs 1-4 performers into a common engineering framework, we present a hypothetical scenario about how our workflow and other CASE technologies could be used by a TA6 performer. Figure 2 illustrates the TA interactions. A TA6 systems engineer, using the CASE integrated tool suite (TA5), plans to design a UAV waypoint navigation system to be cyber-resilient (or perhaps alternatively, re-engineer an existing implementation to improve its cyber resiliency).

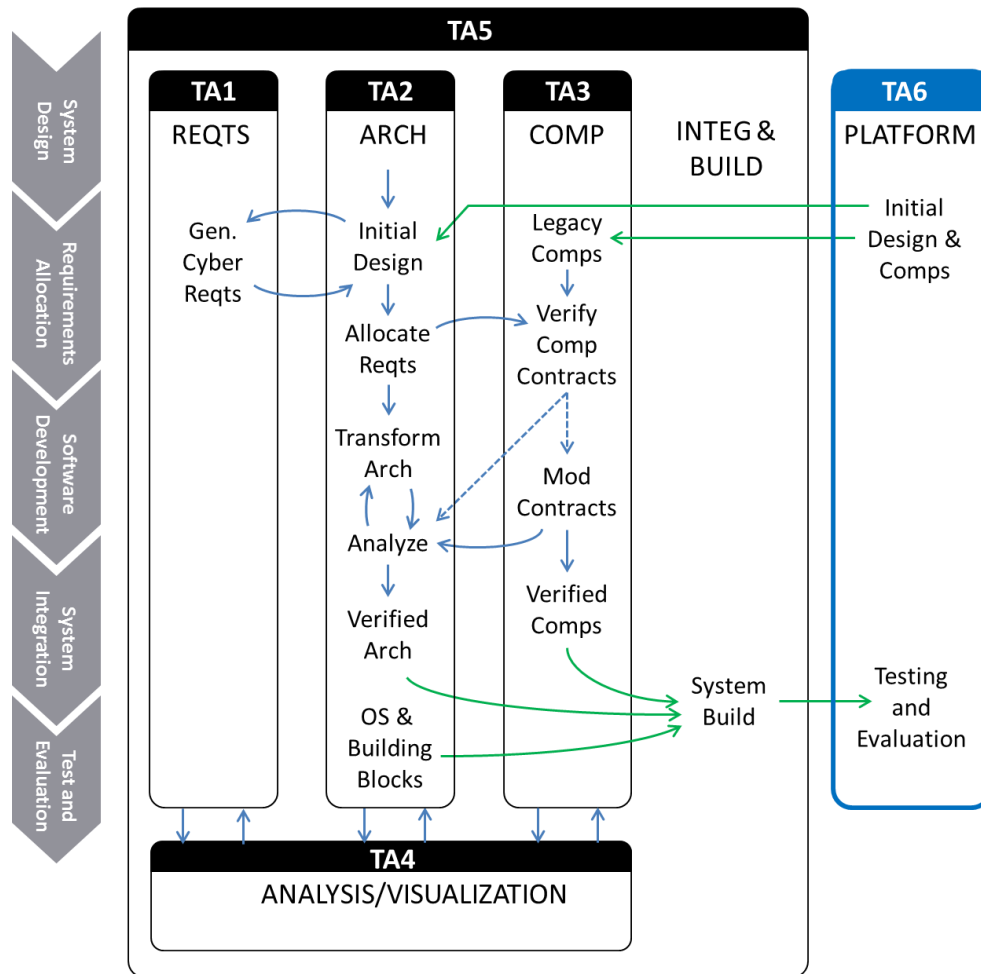


Figure 2. Interactions between TA tools

The TA6 engineer starts with an architecture of the system. The architecture model is created using the TA2 architecture tool. Architecture development is an iterative process. Initially, the system architecture will be driven from a set of high-level functional requirements (some of these requirements are included with our example), or possibly generated from an existing behavioral model (i.e., Simulink). For this example, we have created an AADL model of a simple UAV waypoint navigation system architecture, as pictured in Figure 3.

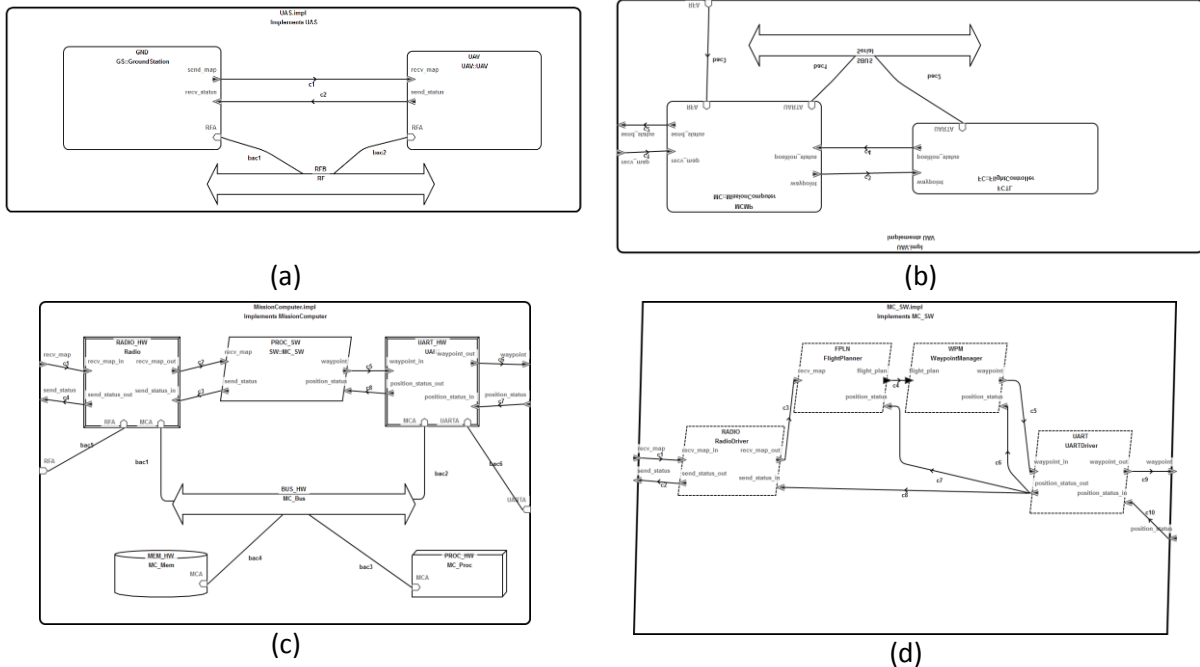


Figure 3. AADL model. (a) Top-level system architecture; (b) UAV; (c) Mission Computer; (d) Software

The architecture (and possibly the functional requirements that drive it) are then passed to TA1 tools. The TA1 tools perform analyses that lead to the generation of cyber requirements. The cyber requirements are consumed by TA2 tools, and the system architecture is modified in response. This interaction between TA1 and TA2 may consist of multiple iterations until the refined architecture no longer results in generation of new cyber requirements from TA1.

In the case where there are legacy software components that will be used in the implementation of this system, the TA2 design tool will determine which TA1 requirements are allocated to these legacy components and TA3 tools will perform adaptations to the code and analyses that verify the code satisfies the TA1 cyber requirements. In our example, the Waypoint Manager (source code provided) is one such legacy component.

The underlying analysis engines that TAs 1-3 utilize will be provided by TA4. These engines will provide feedback in the same semantic context that is being used at the design level.

TAs 1-4 tools will be integrated into a holistic development framework (TA5) that will facilitate tool interoperability and present the TA6 engineer with a useful cyber resilient systems development interface. The TA5 tool will enable the TA6 engineer to perform analyses on the system level, generate design assurance cases, and provide the appropriate mechanisms for building the system.

Demonstration of the TA2 Architecture Tool

Background

To demonstrate our current status and progress on the project, we present a scenario using the simple example described above. We start with an AADL architectural model of the UAS, as well as a set of system requirements that were used to derive the model (see Table 1). To ensure our model satisfies

the requirements, we run the AGREE analysis tool and confirm that each requirement passes. More information on AGREE can be found at:

<http://loonwerks.com/tools/agree.html>

Requirement	Applies to
The Flight Planner shall receive an authenticated command from the Ground Station.	Software
The Flight Planner shall generate a valid mission.	Software
The Waypoint Manager shall output a well-formed mission window.	Software
A CRC shall be appended to the message to determine message correctness.	Software
The Mission Computer shall only accept authenticated commands from the Ground Station	Mission Computer
The Mission Computer shall output a valid mission window to the Flight Controller	Mission Computer

Table 1. UAS Requirements

Because there may be gaps in our requirement specification, especially with respect to cyber-security, we call the TA1 tool to analyze the model and provide additional cyber requirements. For our demo, we assume the TA1 tool generated the following cyber requirement:

The Flight Planner shall receive a well-formed command from the Ground Station.

Although we initially designed the system to reject unauthenticated messages, we did not consider malformed messages. Adding this requirement as an AGREE contract will cause the AGREE analysis to fail because there is nothing in the model that addresses well-formedness.

To address this, we add a special kind of component called a *filter* that enables us to specify the structure of acceptable messages, and the corresponding implementation will prevent malformed messages from reaching their destination. The TA2 Architecture tool will provide a mechanism for inserting the filter component into the desired location on the communication pathway. The AADL filter component already has the necessary AGREE guarantee embedded that will satisfy the requirement. After adding the filter to the model, we can run the AGREE analysis again and verify that all requirements pass.

When we are convinced that the architecture satisfies the requirements, we can integrate and build the system.

Finally, we can generate an assurance case using the Resolute tool to give ourselves confidence in the system design and implementation. The assurance case may contain information and artifacts that are not easily expressed as formal requirements. More information on Resolute can be found at:

<http://loonwerks.com/tools/resolute.html>

Step-by-step Demonstration

All necessary tools and files are provided on the VM. To start, click on the AADL OSATE icon on the Desktop. This launches OSATE, a tool for architecture modeling and analysis. OSATE runs on the eclipse

framework, which we take advantage of to host other design and analysis tools in our development toolchain.

The OSATE workbench should appear as in Figure 4. The Navigator pane on the left-hand side provides a view of the open project. The main editor window displays file contents including text and graphical models. The Outline pane on the right-hand side highlights model components, and the pane on the bottom displays tool outputs.

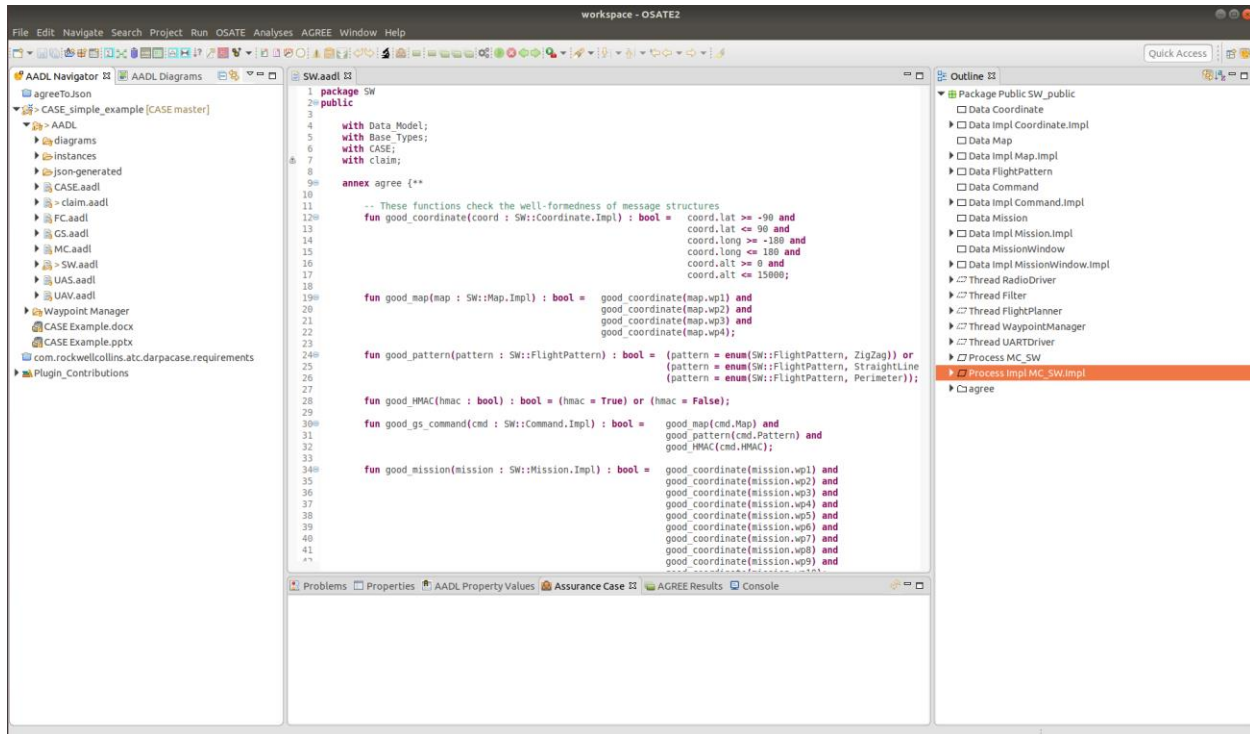


Figure 4. OSATE Architecture Analysis and Design Language tool.

Upon opening the tool, the SW.aadl file should already be open in the editor. If not, expand the CASE_simple_example/AADL folders in the Navigation pane and double-click on the SW.aadl file. The other files in the navigation pane correspond to other systems in our UAS architecture. For example, MC.aadl describes the architecture for the Mission Computer. For a graphical view of any of the systems, right-click on the system in the navigation pane and select “Open Diagram” (or “Create Diagram...” if one does not already exist).

Because our tools are not yet fully automated, this demo will require some manual modification to the model. We will focus only on the Software subsystem of the Mission Computer. The Software subsystem includes all the software components (represented as AADL threads) the Mission Computer requires for reading in a mission command from the Ground Station and generating a mission window for the Flight Controller. The Software subsystem model also defines the data structures consumed and provided by each software component.

Each thread also has associated AGREE contracts specified, which provide guarantees on their output, assuming specific input. The software implementation (MC_SW.Impl) specifies how each component is connected.

Note that several lines have been commented out. As we walk through the example, we will toggle these comments to reflect architecture changes in response to the new TA1 cyber requirement. A comment in AADL has the form:

-- This is a comment

An easy way to toggle comments is to select the desired line(s) of code in the editor and press 'ctrl-/'. This will have the effect of commenting uncommented lines or uncommenting commented lines. Note that selecting both commented and uncommented lines and pressing 'ctrl-/' will produce undesired results.

To start, we have the initial version of the architecture, along with each component's AGREE contracts, as depicted in Figure 5. This figure corresponds to a simplified version of the SW.aadl model (the position status communication pathway has been omitted for clarity).

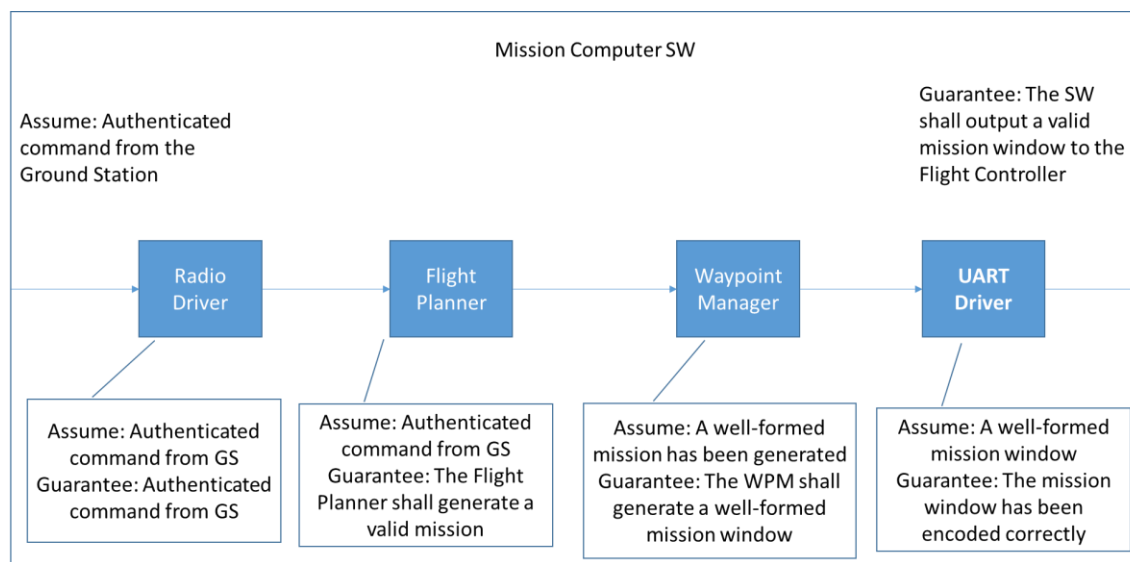


Figure 5. Initial SW Architecture.

If we look at one of the components, for example the Waypoint Manager, its specification (as shown in Figure 6) describes its data ports and associated message types, as well as the AGREE contract it must satisfy.

```

194 thread WaypointManager
195     features
196         flight_plan: in data port Mission.Impl;
197         waypoint: out event data port MissionWindow.Impl;
198         position_status: in event data port Coordinate.Impl;
199
200     annex agree {**
201         assume "The Waypoint Manager shall receive a well-formed mission" : SW.good_mission(flight_plan);
202         guarantee "The Waypoint Manager shall output a well-formed mission window." : SW.good_mission_window(waypoint);
203     **};
204
205 end WaypointManager;

```

Figure 6. Waypoint Manager.

In this case, the AGREE contract states that assuming the Waypoint Manager receives a well-formed mission as input, it will generate a well-formed mission window for the Flight Controller.

The top-level assumption and guarantee in the upper corners of Figure 5 states that the Mission Computer Software will guarantee the generation of a valid mission window, assuming it receives an authenticated command from the Ground Station. The guarantee is only possible due to the guarantee of the UART driver. In turn, the UART driver assumes it will receive a valid mission window, which will only be the case if the component proceeding it on the communication pathway can guarantee it will produce a valid mission window. If a component cannot guarantee a property that the successive component on the communication pathway assumes, AGREE will catch the property violation.

In order to demonstrate that our initial architecture satisfies the requirements listed in Table 1, we run AGREE. To do this, select the “Process Impl MC_SW.Impl” item in the Outline pane and then select AGREE → Verify Single Layer from the OSATE menu (see Figure 7). The output will appear in the Output pane at the bottom of the window (Figure 8). Note that the green check marks next to each item indicate that the properties are valid.

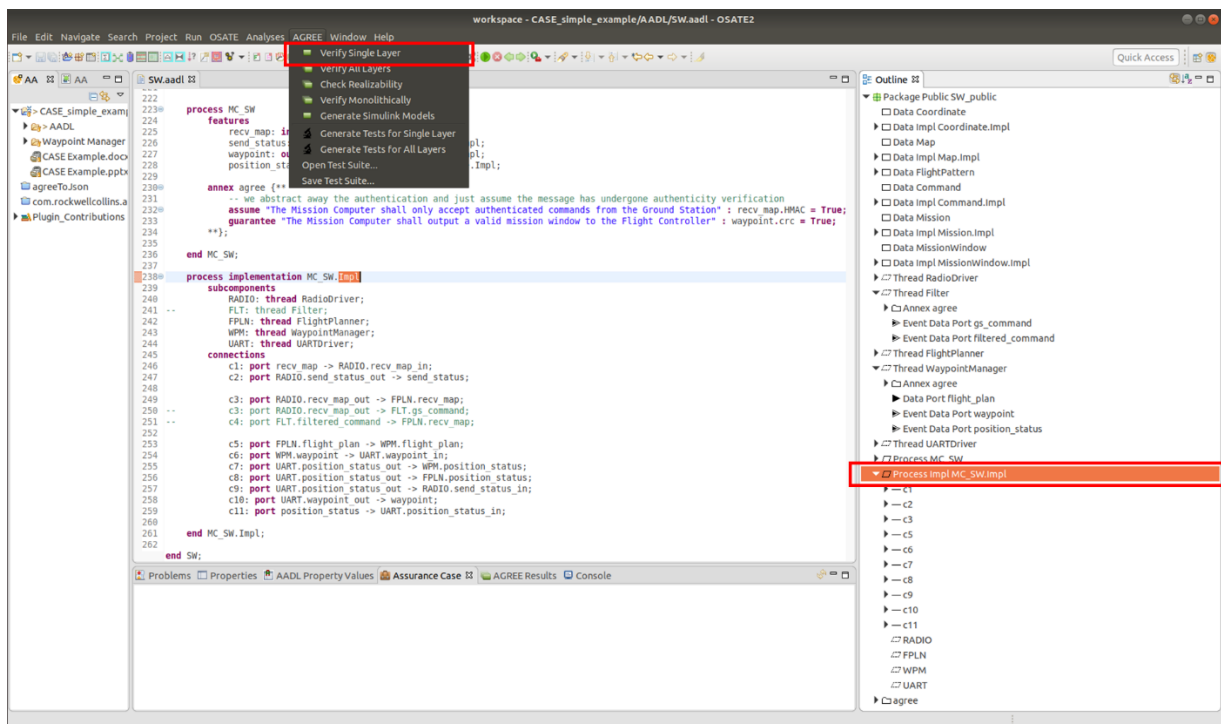


Figure 7. Running AGREE.

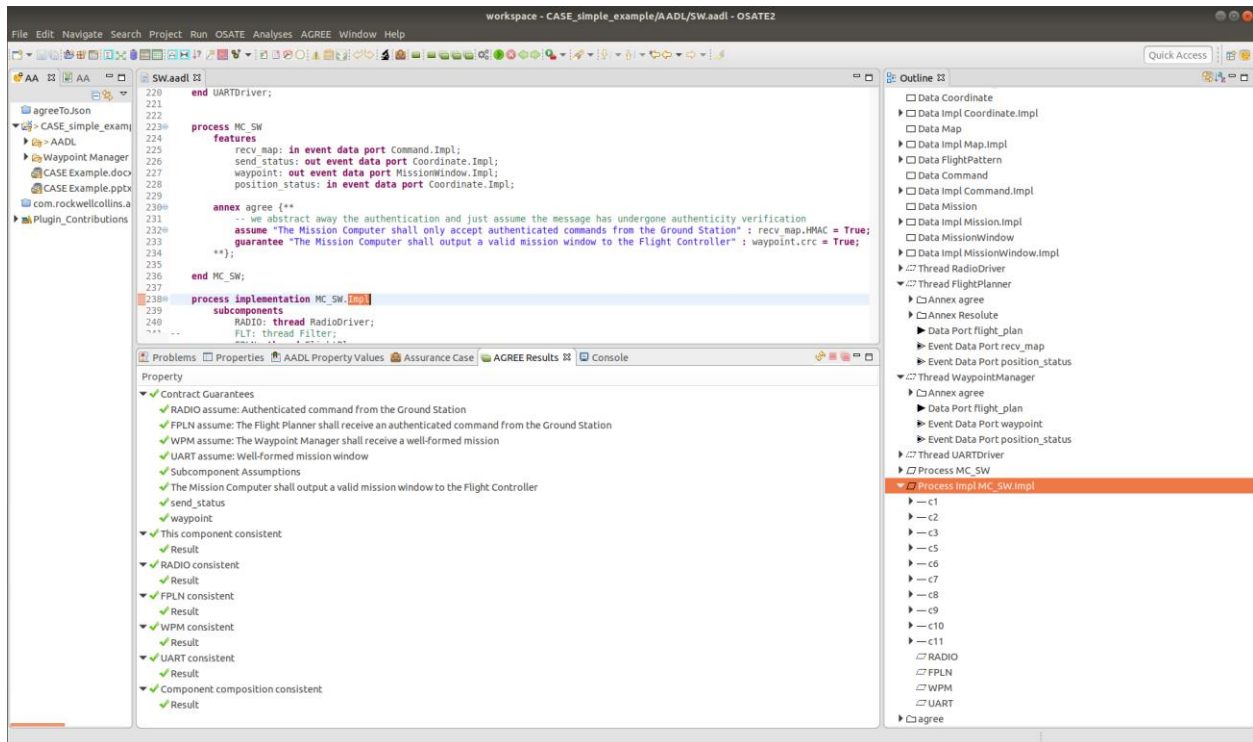


Figure 8. AGREE output.

Now suppose we receive a new well-formedness requirement from TA1. In response, we will add an additional AGREE contract to the Flight Planner. Recall that this new requirement specifies that the Flight Planner shall only receive well-formed messages from the ground station. To incorporate this into the model, we must add a new AGREE assume statement. The assume statement is already present in the model (line 183), but has been commented out. Toggle the comment so the Flight Planner looks like Figure 9 (the lines that remain commented will be addressed later in the demo). We've now declared that the Flight Planner will only guarantee that it generates a valid mission if the message it receives from the Ground Station is both authenticated *and* well-formed. For a message to be well-formed, it must consist of the correct fields, and data values must be in the correct ranges.

```

171
172 thread FlightPlanner
173   features
174     rcv_map: in event data port Command.Impl;
175     flight_plan: out data port Mission.Impl;
176     position_status: in event data port Coordinate.Impl;
177
178   -- properties
179   -- CASE::AGREE_PROPERTIES_PASSED => ("good_gs_command");
180
181   annex agree {**
182     assume "The Flight Planner shall receive an authenticated command from the Ground Station" : rcv_map.HMAC = True;
183     assume "The Flight Planner shall receive a well-formed command from the Ground Station" : SW.good_gs_command(rcv_map);
184     guarantee "The Flight Planner shall generate a valid mission" : SW.good_mission(flight_plan);
185   **};
186
187   -- annex Resolute {**
188   --   prove (well_formed(this, "good_gs_command"))
189   -- **};
190
191 end FlightPlanner;
192

```

Figure 9. Flight Planner with new assume statement.

If we save the model and run AGREE again (remember that the “Process Impl MC_SW.Impl” item in the Outline pane must be selected), we will observe the results shown in Figure 10. Note that the red exclamation marks indicate that the property failed.

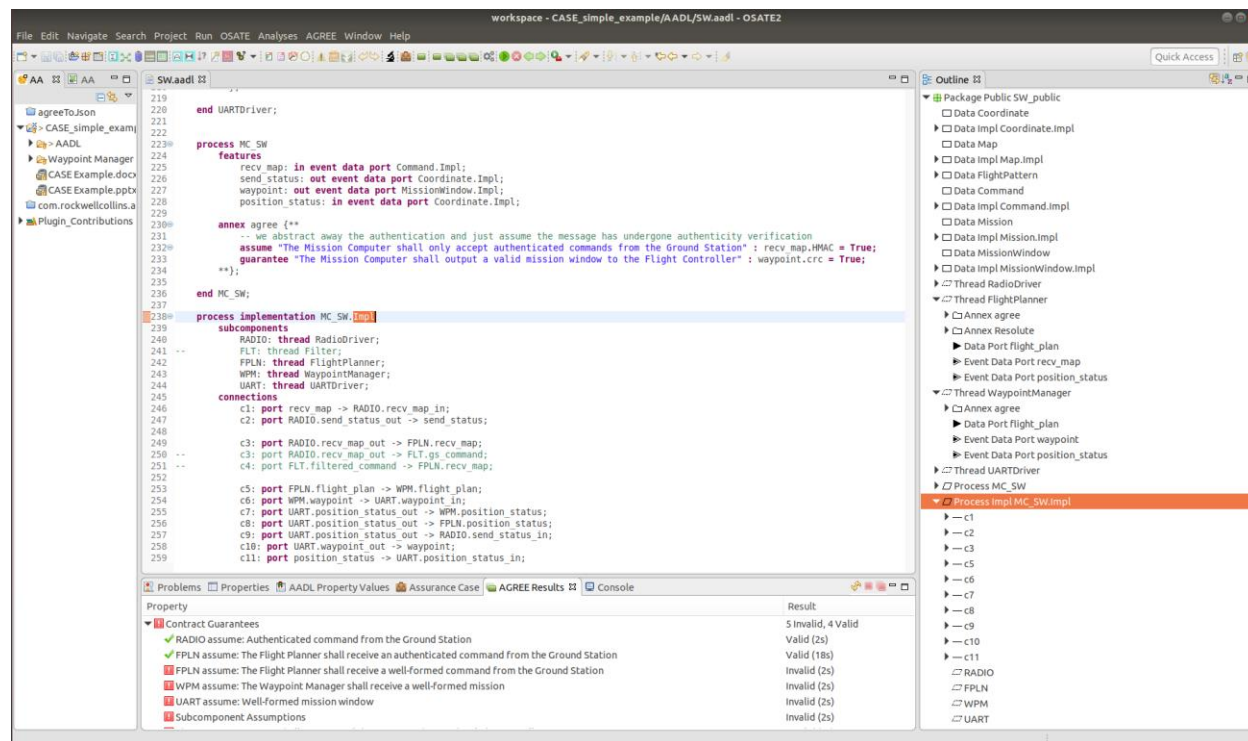


Figure 10. Failed property in AGREE.

The failure occurred because we have no component in our architecture that assures well-formedness. To address this, we must add a filter component that ensures only well-formed command messages reach the Flight Planner, as depicted in Figure 11. The filter component will include the AGREE guarantee necessary to satisfy the assumption specified by the Flight Planner. In future versions, our Architecture tool will feature automation that will directly insert a filter to satisfy this requirement. For now, we will add the filter manually.

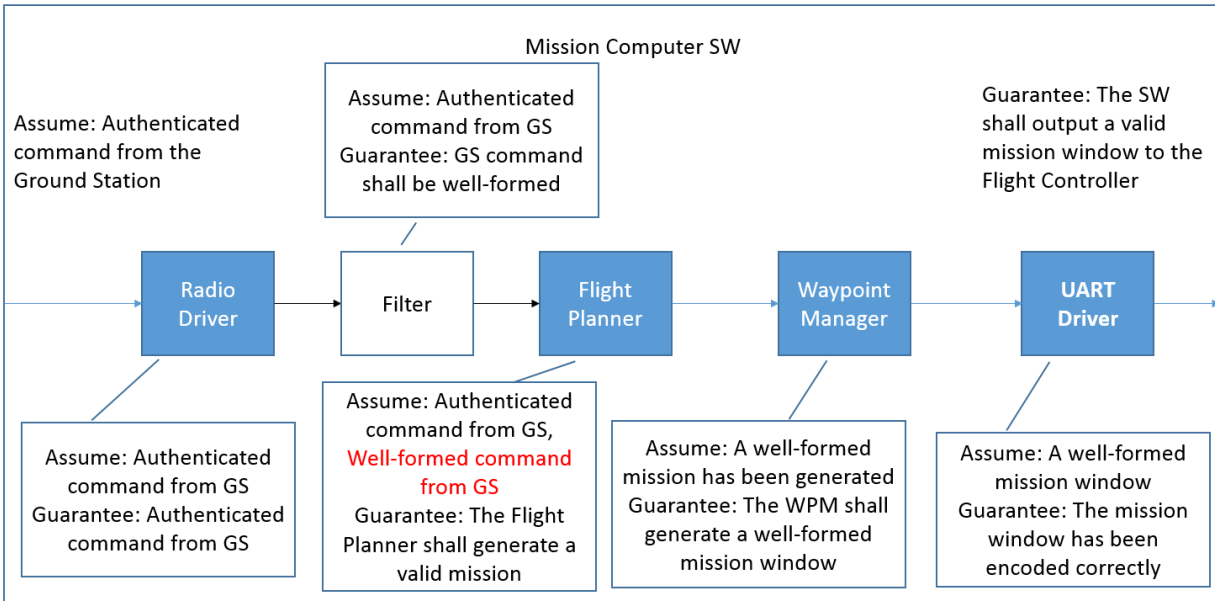


Figure 11. Revised Architecture.

For convenience, we have already included the filter component in the model (although it has been commented out). To wire it in, first uncomment the lines that specify the filter (lines 155-169) so that the code in the editor looks like Figure 12.

```

155 thread Filter
156   features
157     gs_command: in event data port Command.Impl;
158     filtered_command: out event data port Command.Impl;
159   properties
160     CASE::COMP_TYPE => FILTER;
161     CASE::COMP_IMPL => "CAKEML";
162     CASE::COMP_SPEC => "({i{-90,90}}{i{-180,180}}{i{0,15000}}){4}{Z|S|P|T|F}";
163
164 annex agree {**
165   guarantee "The Flight Planner shall receive an authenticated command from the Ground Station" : filtered_command.HMAC = True;
166   guarantee "The Flight Planner shall receive a well-formed command from the Ground Station" : SW.good_gs_command(filtered_command);
167 **};

```

Figure 12. Filter component.

Next, we need to include the filter in the SW Process implementation and wire it correctly so that it sits between the Radio Driver and the Flight Planner on the communication pathway from the Ground Station. To do this, we need to first uncomment the Filter subcomponent instantiation on line 241. Next, we will comment connection c3 on line 249. In our original model, this represented the direct connection between the Radio Driver and the Flight Planner. Finally, we need to uncomment lines 250-251 immediately below. Line 250 connects the Radio Driver to our newly added Filter component, and line 251 connects the Filter to the Flight Planner. Figure 13 shows how the implementation should look once this is done.

```

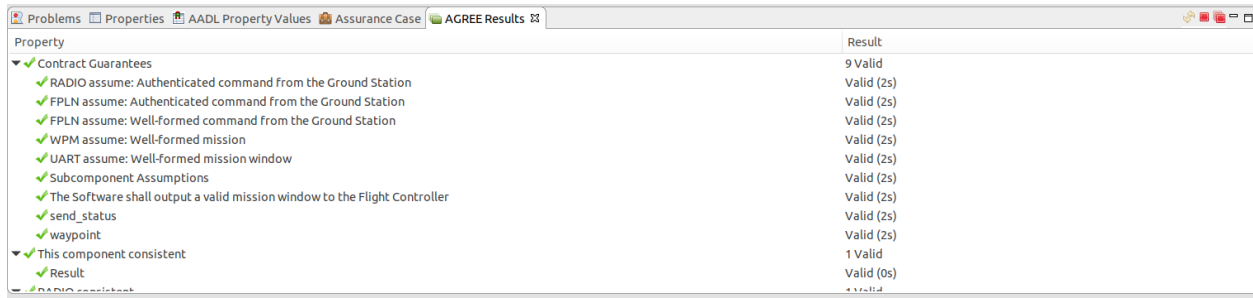
238 | process implementation MC_SW.Impl
239 |   subcomponents
240 |     RADIO: thread RadioDriver;
241 |     --
242 |     FLT: thread Filter;
243 |     FPLN: thread FlightPlanner;
244 |     WPM: thread WaypointManager;
245 |     UART: thread UARTDriver;
246 |   connections
247 |     c1: port recv_map -> RADIO.recv_map_in;
248 |     c2: port RADIO.send_status_out -> send_status;
249 |
250 |     c3: port RADIO.recv_map_out -> FPLN.recv_map;
251 |     --
252 |     c4: port RADIO.recv_map_out -> FLT.gs_command;
253 |     --
254 |     c5: port FLT.filtered_command -> FPLN.recv_map;
255 |
256 |     c6: port FPLN.flight_plan -> WPM.flight_plan;
257 |     c7: port WPM.waypoint -> UART.waypoint_in;
258 |     c8: port UART.position_status_out -> WPM.position_status;
259 |     c9: port UART.position_status_out -> FPLN.position_status;
260 |     c10: port UART.position_status_out -> RADIO.send_status_in;
261 |     c11: port UART.waypoint_out -> waypoint;
262 |     c12: port position_status -> UART.position_status_in;
263 |
264 |   end MC_SW.Impl;
265 | end SW;

```

Figure 13. Filter correctly wired in SW implementation.

Before we re-run the AGREE analysis, some things to note about the Filter component. The component includes some properties that describe its function. These properties are used both by the Resolute analysis as well as by the build process to guarantee the implementation is sound with respect to the requirements. Also note that the second AGREE guarantee provided by the Filter defines what it means for a message to be well-formed, or “good”, by referring to a function defined at the top of the SW.aadl file. The “good” definition is also used later on in the build process.

Make sure to save the model, select the “Process Impl MC_SW.Impl” item in the Outline pane, and run AGREE. The properties should all pass now, and the output should appear as in Figure 14.



Property	Result
Contract Guarantees	9 Valid
✓ RADIO assume: Authenticated command from the Ground Station	Valid (2s)
✓ FPLN assume: Authenticated command from the Ground Station	Valid (2s)
✓ FPLN assume: Well-formed command from the Ground Station	Valid (2s)
✓ WPM assume: Well-formed mission	Valid (2s)
✓ UART assume: Well-formed mission window	Valid (2s)
✓ Subcomponent Assumptions	Valid (2s)
✓ The Software shall output a valid mission window to the Flight Controller	Valid (2s)
✓ send_status	Valid (2s)
✓ waypoint	Valid (2s)
✓ This component consistent	1 Valid
✓ Result	Valid (0s)

Figure 14. AGREE results with Filter component.

In our example, the Filter behavior is defined by a regular expression, as indicated by the filter property COMP_SPEC on line 162 (see Figure 12). The CakeML build tools for the Filter will guarantee that the regular expression is correctly implemented. But we also need to make sure this regular expression correctly implements the AGREE contract for the Filter.

To verify this, make sure the file SW.aadl is open in the OSATE editor and press the gear button on the toolbar (see Figure 15). If the button is disabled, make sure the editor is the active pane. This has the effect of exporting the model to run a proof procedure using PolyML/HOL4.

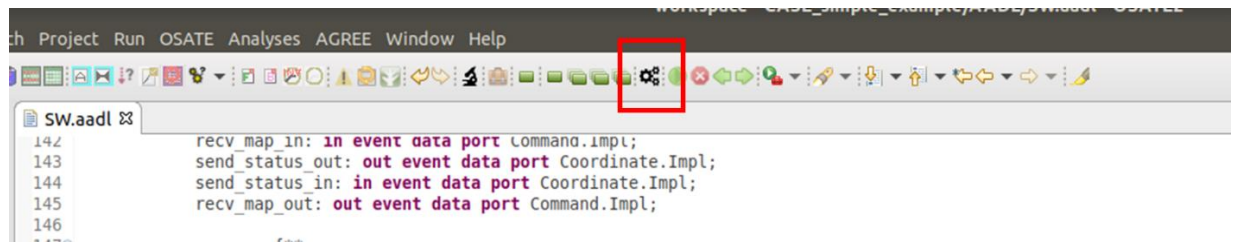


Figure 15. Initiate proof procedure.

A dialog box is displayed to inform us that a proof procedure is attempting to verify that our AGREE claims adhere to the regular expression definition of the filter (Figure 16).

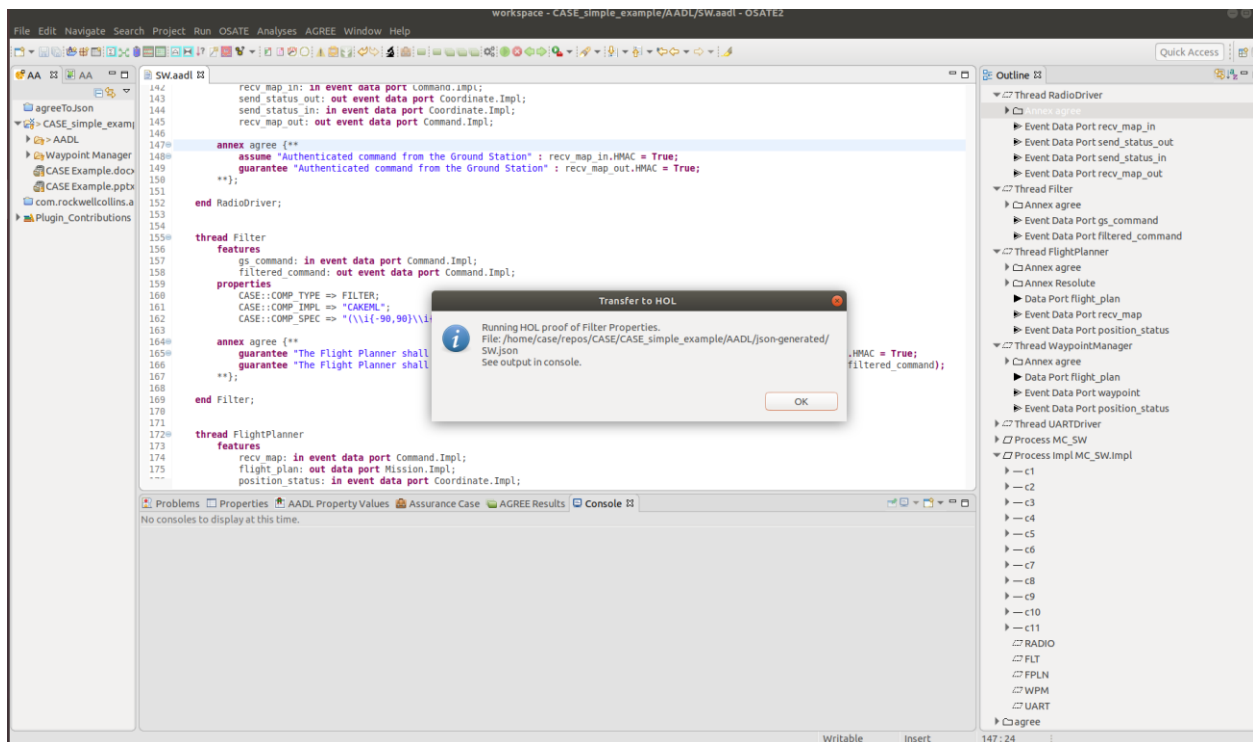


Figure 16. Proof procedure status.

The result of the proof process is displayed in the OSATE console (see Figure 17).

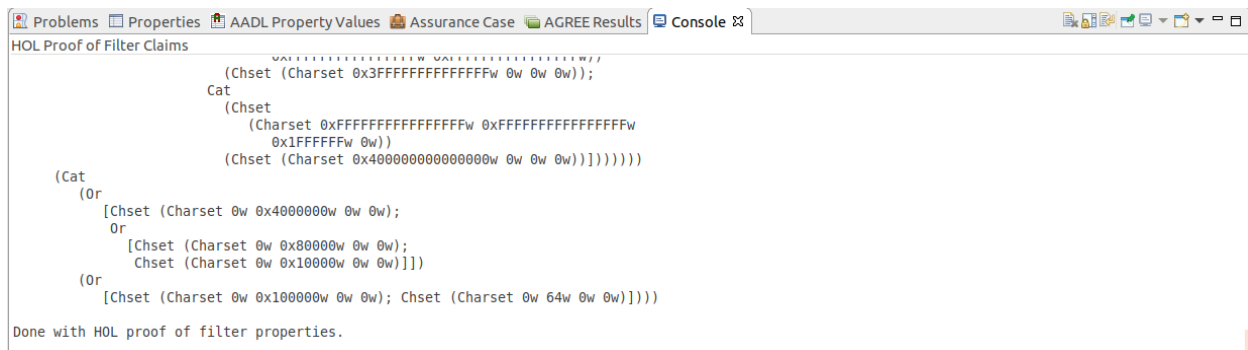


Figure 17. HOL proof of filter claims

The actual proof procedure for verifying that AGREE claims adhere to regular expressions is under active development.

If the proof completes successfully, a certificate is generated, which will be associated with the Filter component in our SW model. This proof certificate will be necessary to build the Resolute assurance case in later stages of the development process.

An assurance case gives us confidence that our development process is sound. The Resolute tool helps us build these assurance cases. In this instance, we may want to show that the steps we took adding the Filter to our design were appropriate with respect to the desired integrity level of the system.

When we add the Filter, we also add a Resolute clause to the Flight Planner component that enables us to demonstrate the Filter will produce the desired behavior. For example, the addition of a filter will not ensure well-formedness if there is another communication pathway to the Flight Planner that does not include a filter. The addition of a filter in our model is also meaningless unless we can show that the corresponding implementation also exhibits the desired behavior.

A Resolute clause has already been added to the Flight Planner component, but is commented out. Uncomment the Properties section on lines 178-179, as well as the Resolute clause on lines 187-189. The Flight Planner thread should now appear as in Figure 18.

```

171
172 thread FlightPlanner
173   features
174     recv_map: in event data port Command.Impl;
175     flight_plan: out data port Mission.Impl;
176     position_status: in event data port Coordinate.Impl;
177
178   properties
179     CASE::AGREE_PROPERTIES_PASSED => ("good_gs_command");
180
181 annex agree {**
182   assume "The Flight Planner shall receive an authenticated command from the Ground Station" : recv_map.HMAC = True;
183   assume "The Flight Planner shall receive a well-formed command from the Ground Station" : SW.good_gs_command(recv_map);
184   guarantee "The Flight Planner shall generate a valid mission" : SW.good_mission(flight_plan);
185 **};
186
187 annex Resolute {**
188   prove (well_formed(this, "good_gs_command"))
189 **};
190
191 end FlightPlanner;
192

```

Figure 18. Resolute clause.

The property on line 179 indicates that the 'good_gs_command' property was checked by AGREE and passed. In future versions of the tool, AGREE will be able to communicate directly with Resolute to indicate which properties it has checked. Because this functionality is not currently implemented we indicate this manually by referencing the requirement in the AADL property. The clause (on line 188) calls on Resolute to generate an assurance case for the new well-formedness cyber requirement. The well_formed() function is defined in the claim.aadl file. For this example, Resolute must show that:

- a) the Filter precedes the Flight Planner on the communication pathway
- b) the Filter cannot be bypassed
- c) the Filter has been implemented by CakeML
- d) the system property has been checked by AGREE

To run Resolute, save the model, then right-click on the "Process Impl MC_SW.Impl" item in the Outline pane and select Resolute from the pop-up menu. The results appear in the Output pane and should resemble Figure 19.

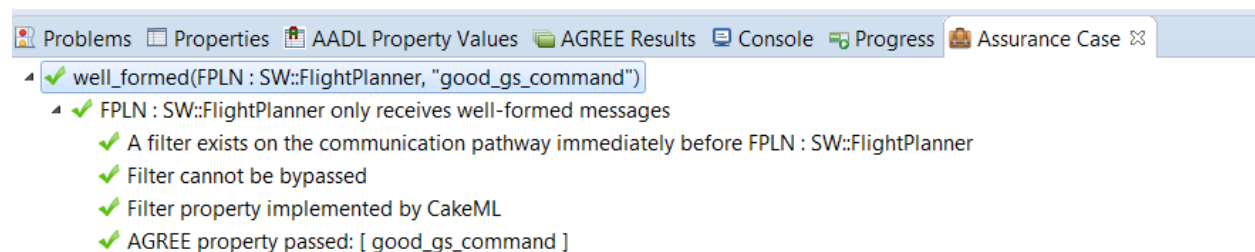


Figure 19. Resolute results.

To see an example of how Resolute will catch an invalid claim, comment line 160 as illustrated in Figure 20, save the model, and re-run Resolute. Note the expected result in the Output pane of the figure.

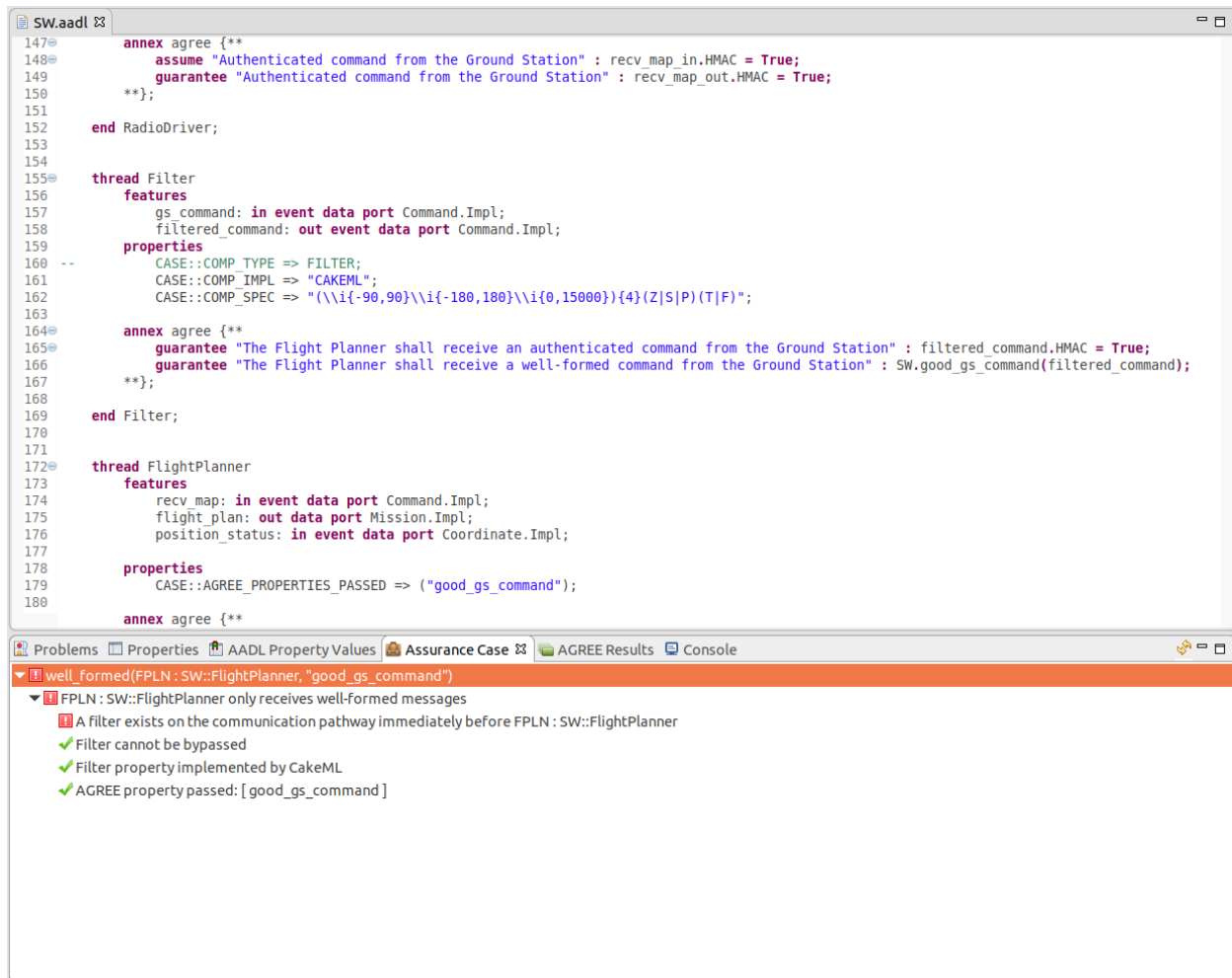


Figure 20. Invalid Resolute claim.

The regular expression that defines the filter is used to generate an executable as well as a certificate stating that the executable adheres to the regular expression specification. Future versions of our tool will be able to synthesize the filter executable directly from the AADL component and associate both the executable and proof certificate with the Filter component via annotations in the AADL model.

To demonstrate the results of the build process, we will run the high-assurance filter in a simple test harness as a CAMkES system, consisting of a producer, the filter, and a consumer, all running on seL4 (Figure 21). The producer emits strings over IPC to the CakeML Filter, which matches them against a regular expression and either forwards them via IPC to the Consumer (if they match), or drops them (if they do not). The CakeML Filter is accompanied by proofs generated as part of the build process. These proofs can then be checked by the HOL4 proof assistant to guarantee that the implementation of regular expression matching is correct.

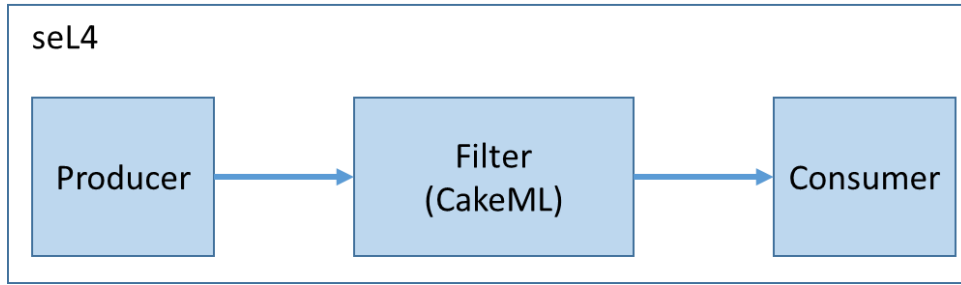


Figure 21. Filter test setup.

A CAMkES system consists of several *components*, communicating via *connections*. In our example system, our three components communicate via two RPC connections, configured in `apps/cakeml-filter/cakeml-filter.camkes`. This `.camkes` file describes the structure of the system in a way that allows the CAMkES tool to generate C "glue code" to join the components together into a running system. Each component is also specified in its own `.camkes` file, which describes the properties of the component (such as whether it has a control thread) and any interfaces it provides or uses.

For general information on CAMkES, please see the documentation at:

<https://sel4.systems/Info/CAMkES/About.pml>

<https://github.com/sel4/camkes-tool/blob/master/docs/index.md>

Detailed information on the filter implementation and execution is located in the following directory:

`~/CASE/cakeml-filter/README.md`

To build and test the filter, first open the message input file:

`~/CASE/cakeml-filter/projects/cakeml/apps/cakeml-filter/components/Producer/producer.c`

This file enables us to input messages that the Producer will attempt to communicate to the Consumer through the Filter (see Figure 22).

```

#include <camkes.h>

int run() {
    server_transfer_string("-50:-180:10000:-90:100:10000:20:010:10000:90:100:10000:Z:T\n"); // Pass
    server_transfer_string("-50:-180:10000:-90:100:10000:20:-010:10000:90:100:10000:P:T\n"); // Pass
    server_transfer_string("-50:-180:10000:-90:100:10000:20:010:10000:90:100:10000:Z:F\n"); // Pass
    server_transfer_string("-500:100:10000:-50:100:10000:-50:100:10000:-50:100:10000:Z:T\n"); // Fail
    server_transfer_string("-50:100:-10000:-50:100:10000:-50:10:10000:-50:100:10000:Z:T\n"); // Fail
    server_transfer_string("-500:100:10000:-50:100:10000:-50:100:10000:-50:100:10000:A:T\n"); // Fail
    server_transfer_string("-500:100:10000:-50:100:10000:-50:100:10000:-50:100:10000:Z:L\n"); // Fail
    server_transfer_string("-500:100:10000:-50:100:10000:-50:100:10000:Z:L\n"); // Fail
    server_transfer_string("-500:100:10000:-50:100:10000:-50:100:10000:-50:100:10000:Z\n"); // Fail
}

```

Figure 22. Producer message file.

We have created several messages to test, and have commented each message to indicate the expected outcome.

Messages are formatted according to the regular expression:

`(-?[0-9]{2}:-?[0-9]{3}:[0-9]{5}:){4}[ZSP]:[TF].*`

which is specified in the file

~/CASE/cakeml-filter/projects/cakeml/apps/cakeml-filter/CMakeLists.txt

Each message consists of four groups of lat:long:alt coordinates, which enclose the UAV surveillance region, a character representing the desired flight pattern (Zig-zag, Straight-Line, or Perimeter), and an authentication bit (Truer or False). Note that the lat:long:alt ranges allowed by the above regular expression are greater than the ranges defined in our AADL model. This was done intentionally to keep this example simple. Also note that the .* is necessary at the end of the regular expression because messages are padded with zeros to realize a constant-size buffer.

Additional messages may be added to the Producer.c.

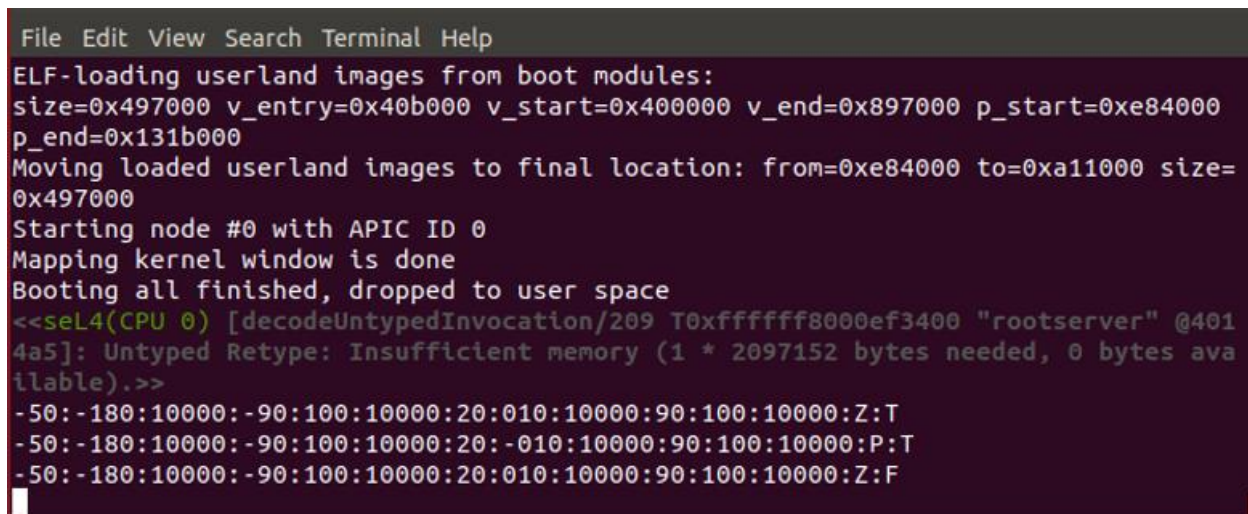
In order to build and run the example, open the terminal and navigate to the cakeml-filter directory by typing

cd ~/CASE/cakeml-filter

Next, type

./build_and_run.sh

The build can take several minutes (if you have made any changes to the regular expression or the test messages in producer.c). The output will consist of the messages that the filter allows to pass to the Consumer. For the input as shown in Figure 22, the output will appear as in Figure 23. The error message directly preceding the message output is a known issue and will be addressed in future versions of the tool.



```
File Edit View Search Terminal Help
ELF-loading userland images from boot modules:
size=0x497000 v_entry=0x40b000 v_start=0x400000 v_end=0x897000 p_start=0xe84000
p_end=0x131b000
Moving loaded userland images to final location: from=0xe84000 to=0xa11000 size=
0x497000
Starting node #0 with APIC ID 0
Mapping kernel window is done
Booting all finished, dropped to user space
<<seL4(CPU 0) [decodeUntypedInvocation/209 T0xffffffff8000ef3400 "rootserver" @401
4a5]: Untyped Retype: Insufficient memory (1 * 2097152 bytes needed, 0 bytes ava
ilable).>>
-50:-180:10000:-90:100:10000:20:010:10000:90:100:10000:Z:T
-50:-180:10000:-90:100:10000:20:-010:10000:90:100:10000:P:T
-50:-180:10000:-90:100:10000:20:010:10000:90:100:10000:Z:F
```

Figure 23. CAMkES filter output.

To exit QEMU once it has completed execution, enter 'ctrl-a' followed by 'x'.

Note that if you make changes to the producer.c file, you must delete the ~/CASE/cakeml-filter/build directory by typing

rm -r ./build