CASE Tool Demo

Tool Assessment 2

Collins Aerospace – 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.

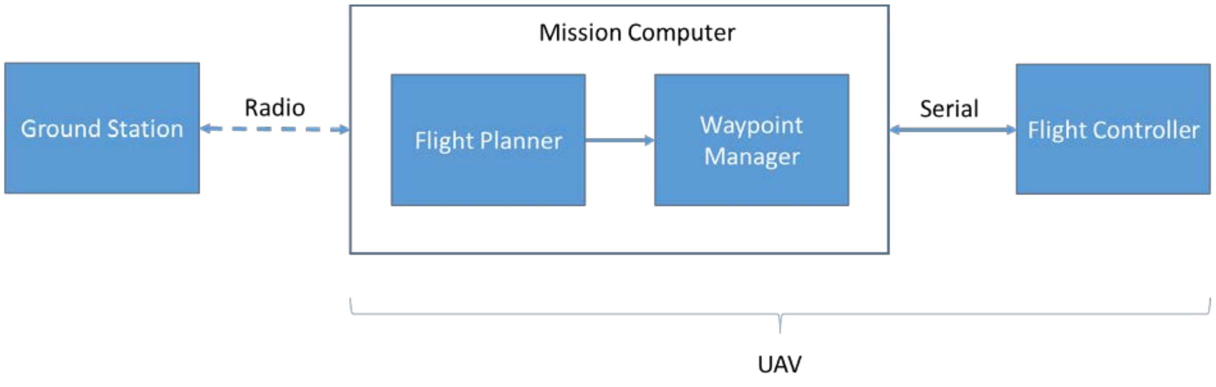


Figure 1. Waypoint Navigation System

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).

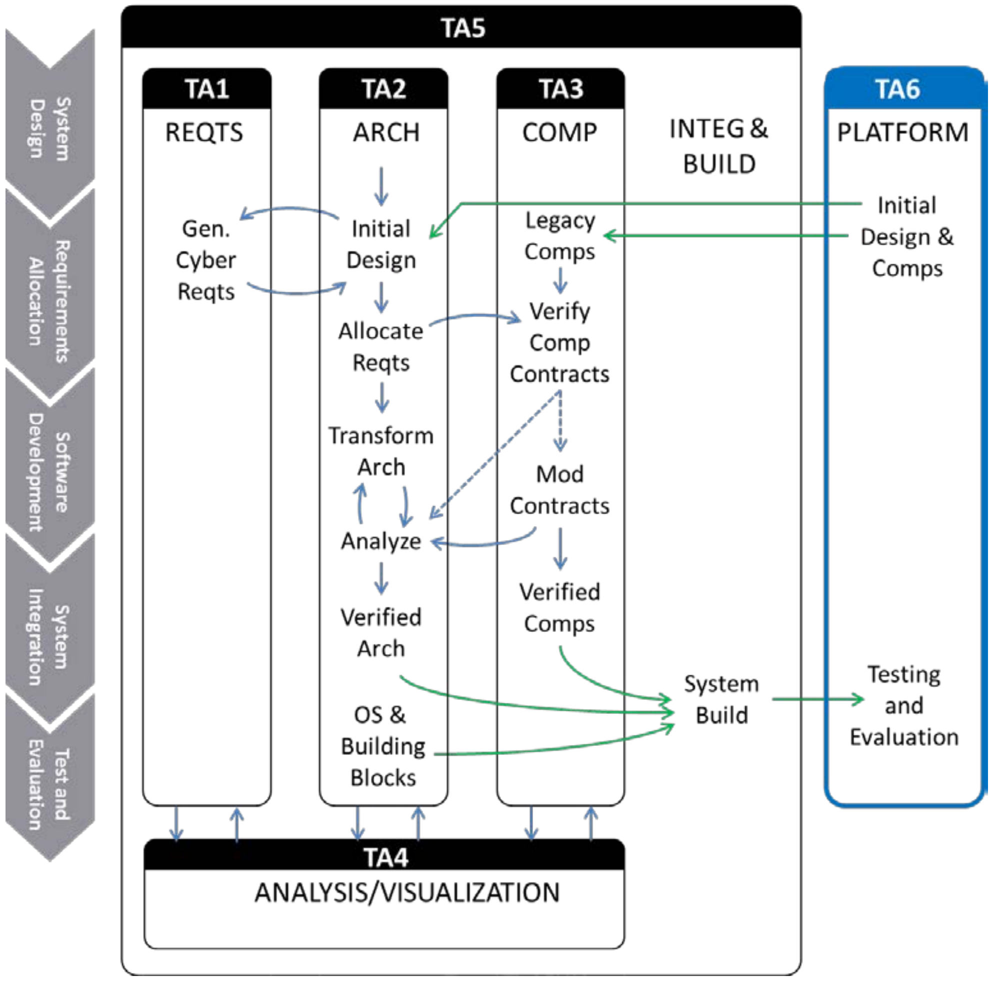


Figure 2. Interactions between TA tools

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*

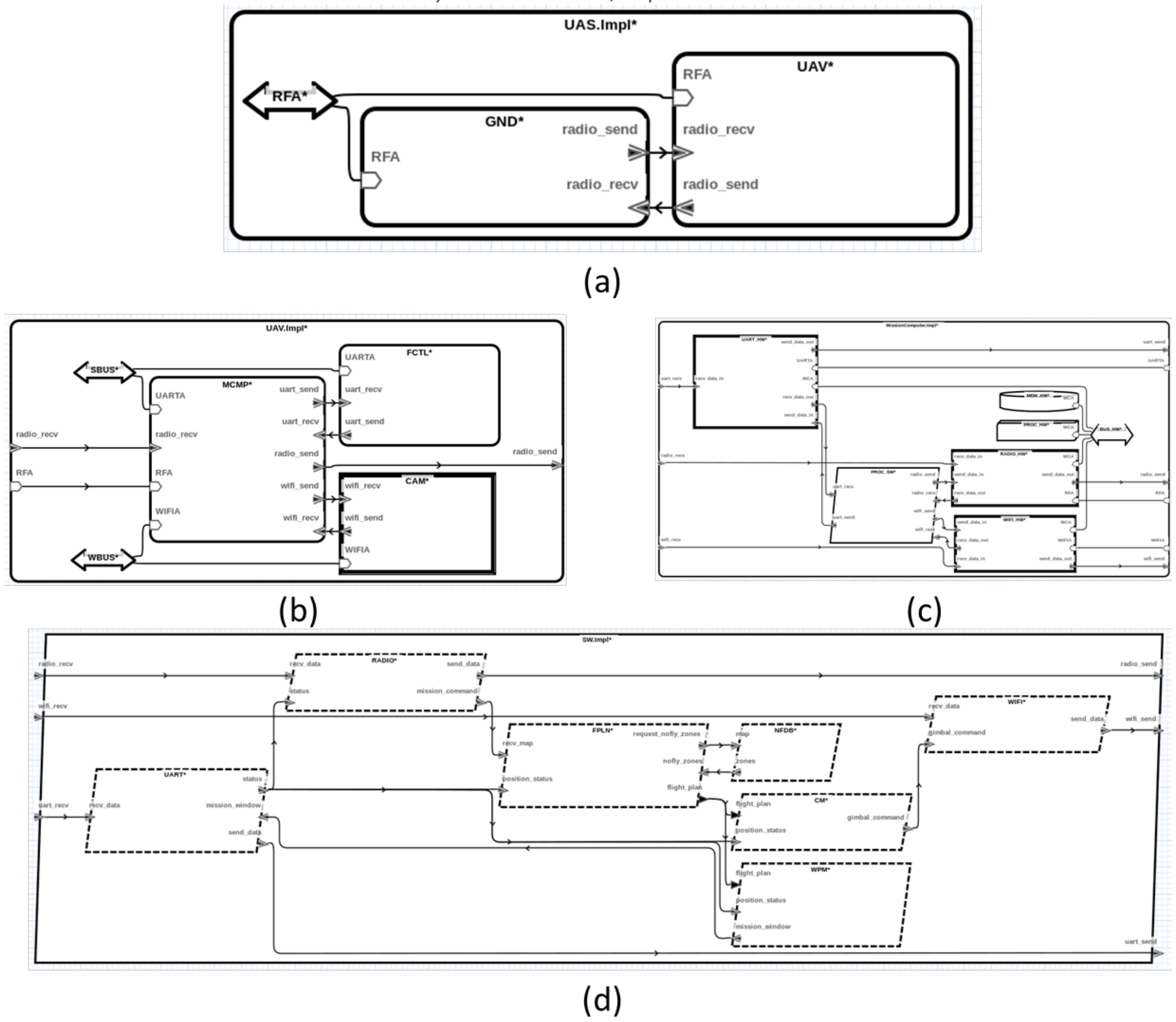


Figure 3. This model has been extended in a number of ways since the July 2018 evaluation. It includes a more detailed description of the computing hardware. Components have been updated with properties indicating their security needs (for confidentiality, integrity, and availability). Information has been added about communication modalities used by the buses in the model.

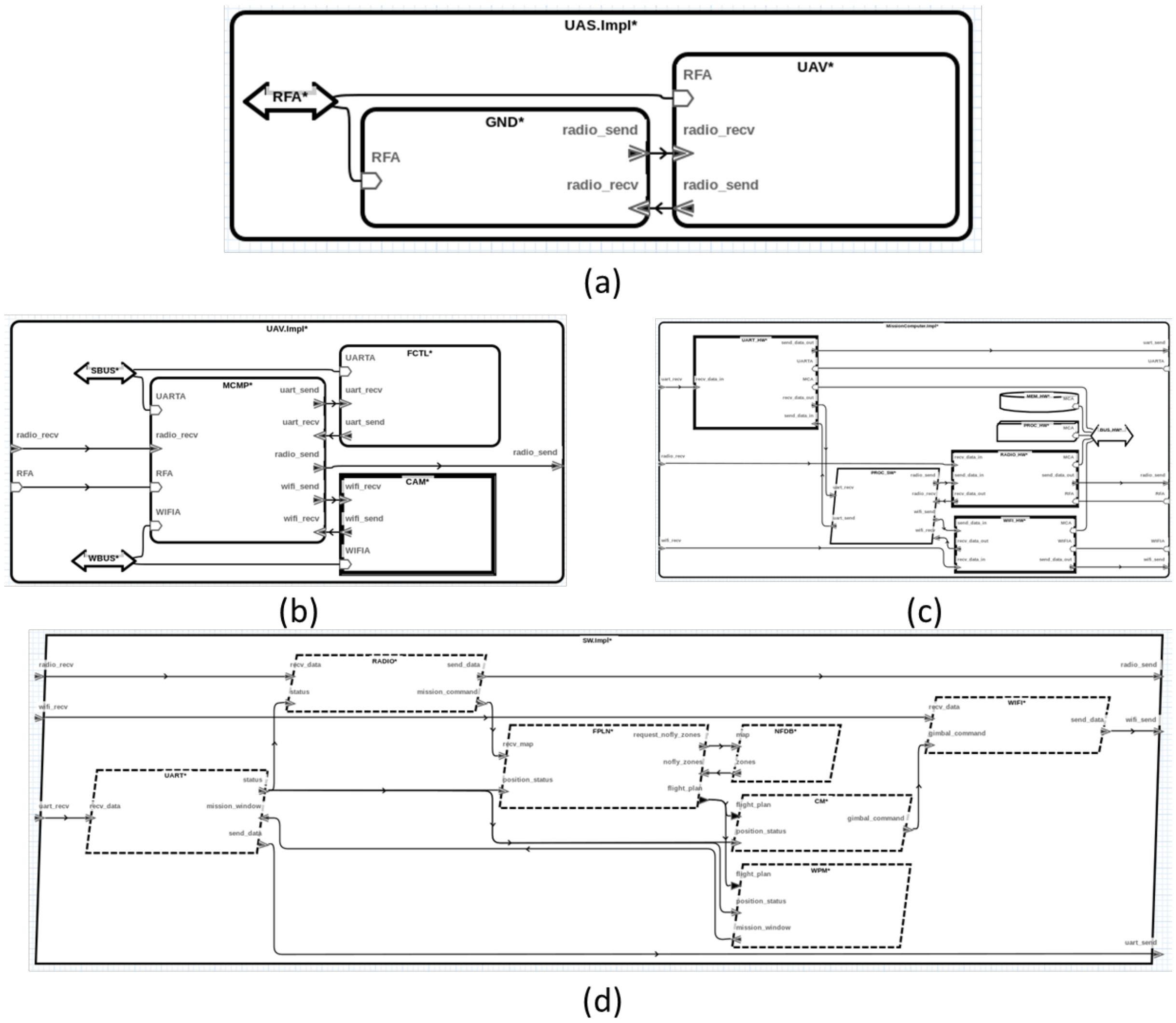


Figure 3. AADL model. (a) Top-level system architecture; (b) UAV; (c) Mission Computer; (d) Software

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 formal 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.

Overview of the Demonstration

Figure 4 shows an overview of the software provided in the demonstration. The yellow components are the tools and models associated with the TA2 cyber-resiliency transformations. The blue components are the tools and models demonstrating the high-assurance filter inserted by the TA2 tools. The orange component is a demonstration of a new remote attestation component that will be inserted by the TA2 tools. The green components are the tools and models demonstrating our initial TA5 System Build capability.

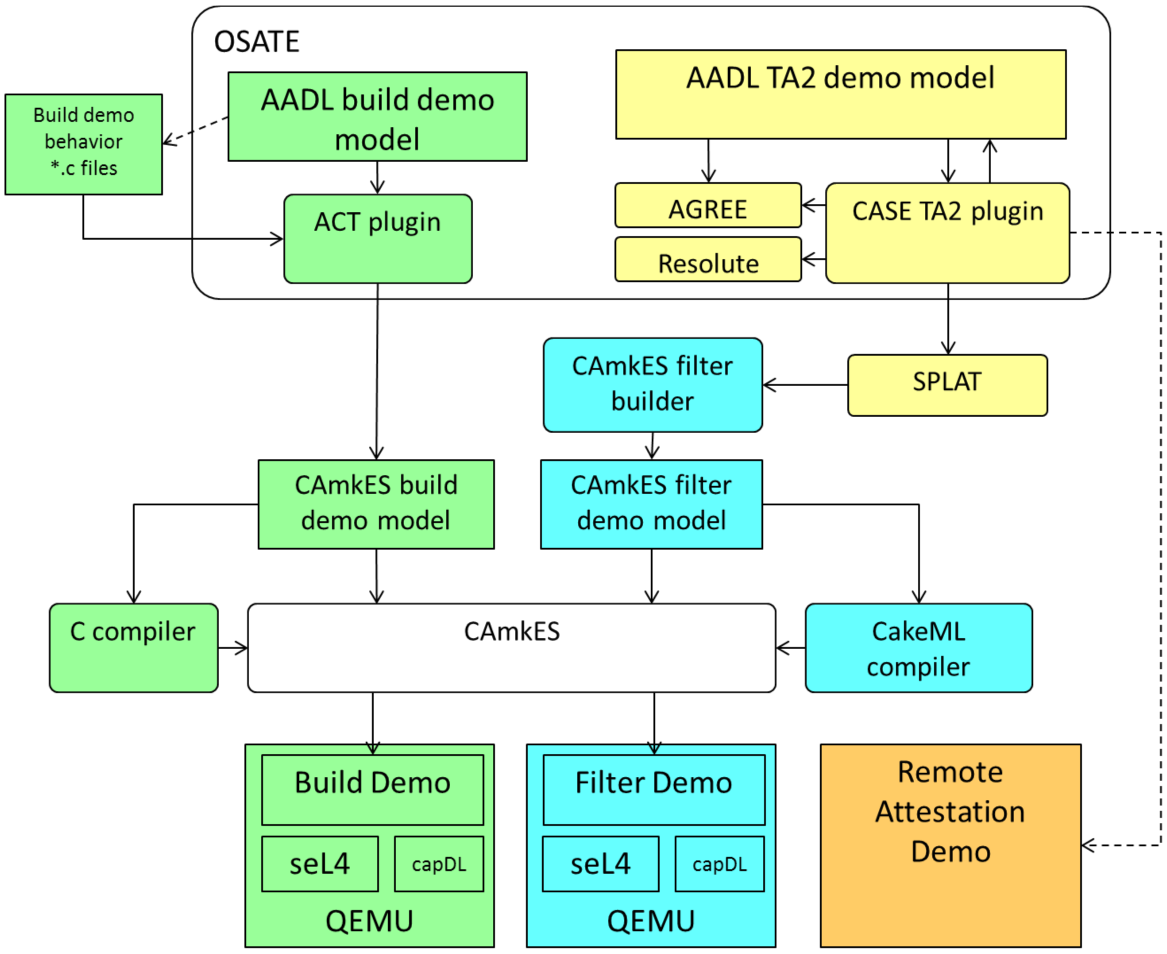


Figure 4. Overview of the Demonstration software

At this point, the AADL-to-CAmkES system build software only supports a subset of our AADL models. In phase 2 releases, this will be expanded to provide full coverage of the AADL models including the high-assurance filter and attestation components. This will result in a single executable built from a single AADL model, as shown in *Figure 5*.

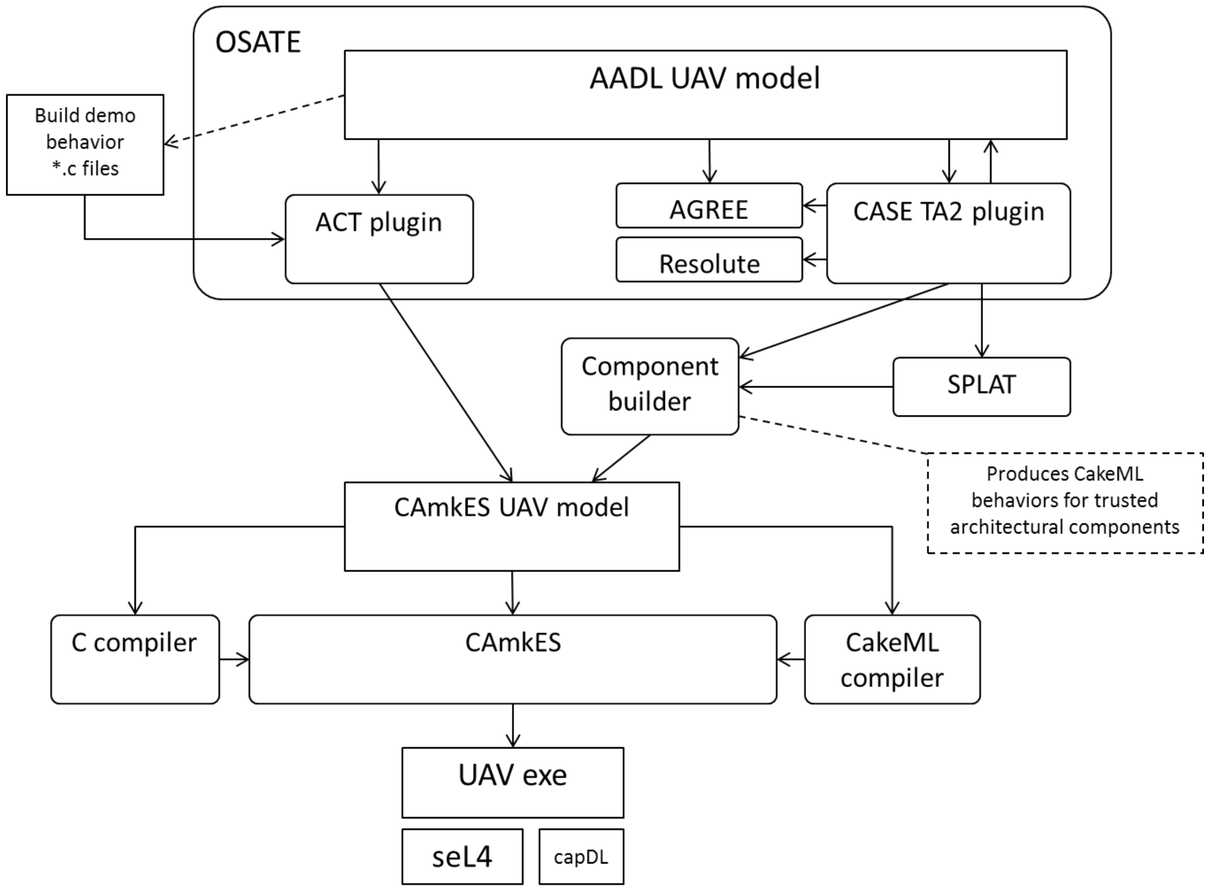


Figure 5. Future integrated tool architecture.

Demonstration of the TA2 Architecture Tool

Overview

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 a valid (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 process messages from the Ground Station intended for it	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 requirements:

The Flight Planner shall receive messages from a trusted Ground Station.

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 these requirements as AGREE contracts will cause the AGREE analysis to fail because there is nothing in the model that addresses either trustworthiness or well-formedness.

To address this, we perform transformations on the model to add special CASE components (such as an *attestation manager* and a *filter*) that enable us to specify the structure of acceptable messages, and the corresponding implementation will prevent malformed or untrusted messages from reaching their destination. The TA2 Architecture tool will provide a mechanism for inserting the CASE components into the desired location in the architecture. The CASE components already have the necessary AGREE guarantees embedded that will satisfy the requirement. After adding the components 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:

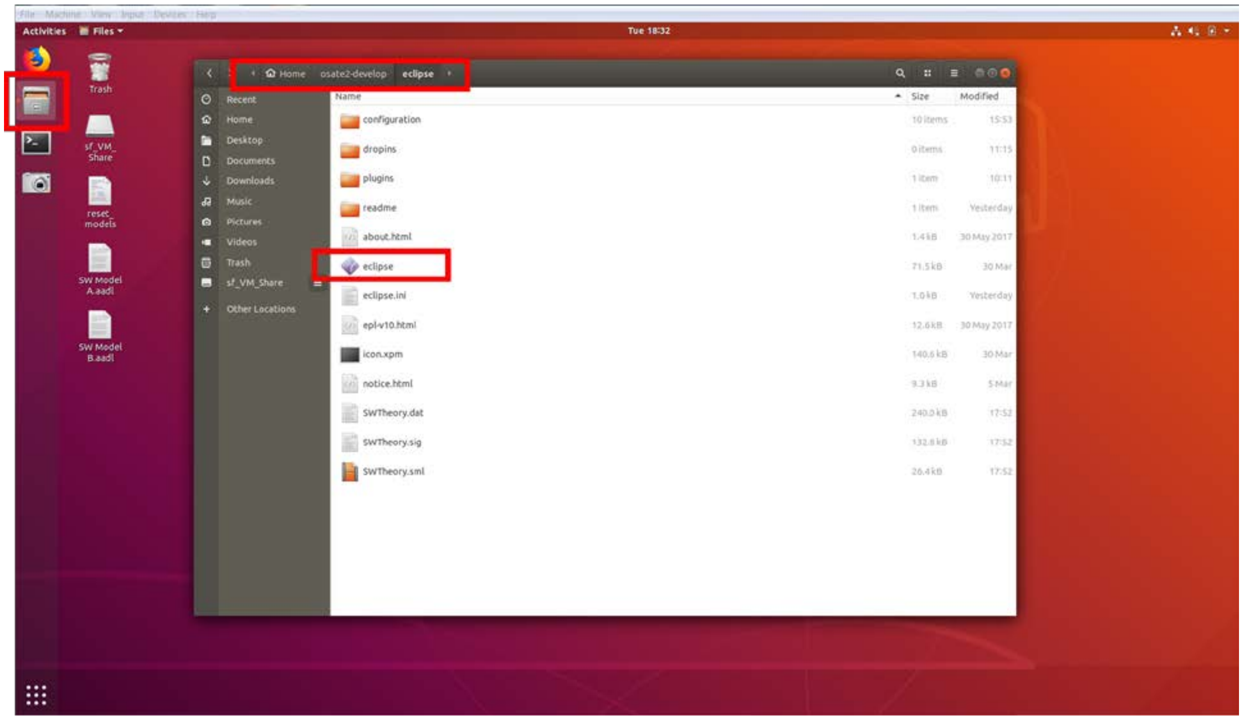
http://loonwerks.com/tools/resolute.html

Step-by-step Demonstration

NOTE: All necessary tools and files are provided on the VM. The OS password is “aerospace”.

NOTE: If at any time you wish to start over, the reset_models script will reset the AADL files in the CASE_Simple_Example_V2 directory to their initial state. In addition, copies of the Initial (Model A) and Resilient (Model B) are on the Desktop.

To start the demo, open the files explorer on the left icon bar, navigate to Home/osate2-develop/eclipse/, and launch Eclipse by clicking on the eclipse (see Figure 6). The OSATE project source, including the Collins CASE contribution will be loaded by default. Launch OSATE by clicking on the Run icon (shown in Figure 7). In future versions, building OSATE will not be necessary, and CASE tools will be integrated directly with the standalone OSATE application. OSATE is a tool for architecture modeling and analysis. Because OSATE runs on the eclipse framework, we use this for development of our CASE tools to take advantage of its tight integration capabilities via plugins.



Name	Size	Modified
configuration	10 items	15:53
dropins	0 items	11:15
plugins	1 item	10:11
readme	1 item	Yesterday
about.html	1.4 kB	30 May 2017
eclipse	71.5 kB	30 Mar
eclipse.ini	1.0 kB	Yesterday
epl-v10.html	12.6 kB	30 May 2017
icon.xpm	140.6 kB	30 Mar
notice.html	9.3 kB	5 Mar
SWTheory.dat	240.0 kB	17:52
SWTheory.sig	132.8 kB	17:52
SWTheory.sml	26.4 kB	17:52

Figure 6. Launching Eclipse development environment.

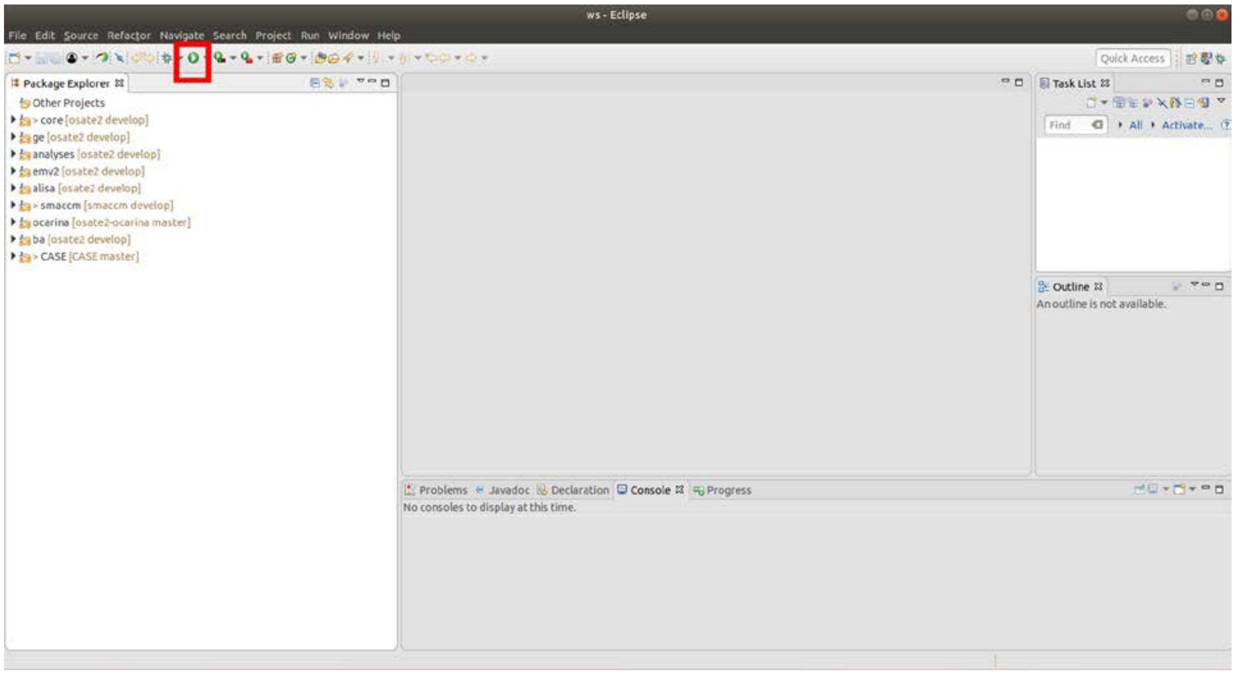
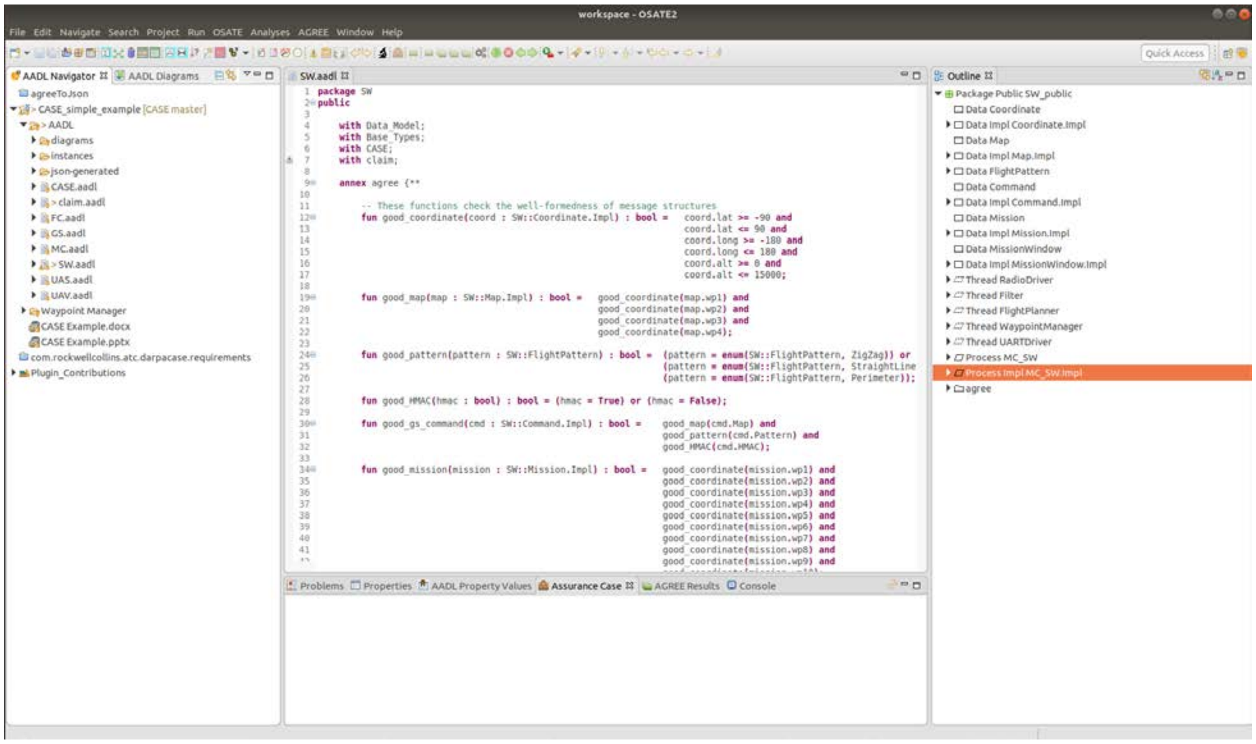


Figure 7. Run OSATE from Eclipse.

Once launched, the OSATE workbench should appear as in Figure 8. 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.



```
package SW
public

    with Data_Model;
    with Base_Types;
    with CASE;
    with claim;

    annex agree {**

        -- These functions check the well-formedness of message structures
        fun good_coordinate(coord : SW::Coordinate.Impl) : bool =   coord.lat >= -90 and
                                                                    coord.lat <= 90 and
                                                                    coord.long >= -180 and
                                                                    coord.long <= 180 and
                                                                    coord.alt >= 0 and
                                                                    coord.alt <= 15000;

        fun good_map(map : SW::Map.Impl) : bool =   good_coordinate(map.wp1) and
                                                    good_coordinate(map.wp2) and
                                                    good_coordinate(map.wp3) and
                                                    good_coordinate(map.wp4);

        fun good_pattern(pattern : SW::FlightPattern) : bool =  (pattern = enum(SW::FlightPattern, ZigZag)) or
                                                                (pattern = enum(SW::FlightPattern, StraightLine
                                                                (pattern = enum(SW::FlightPattern, Perimeter));

        fun good_HMAC(hmac : bool) : bool = (hmac = True) or (hmac = False);

        fun good_gs_command(cmd : SW::Command.Impl) : bool =    good_map(cmd.Map) and
                                                                good_pattern(cmd.Pattern) and
                                                                good_HMAC(cmd.HMAC);

        fun good_mission(mission : SW::Mission.Impl) : bool =   good_coordinate(mission.wp1) and
                                                                good_coordinate(mission.wp2) and
                                                                good_coordinate(mission.wp3) and
                                                                good_coordinate(mission.wp4) and
                                                                good_coordinate(mission.wp5) and
                                                                good_coordinate(mission.wp6) and
                                                                good_coordinate(mission.wp7) and
                                                                good_coordinate(mission.wp8) and
                                                                good_coordinate(mission.wp9) and
```

Figure 8. 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_V2 folder 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). Note that AADL models are represented both declaratively using a text editor, or graphically. For this demonstration we operate solely in the textual environment, however, we do plan on implementing the functionality for performing the same activities in the graphical environment in the future.

The demo focuses on the Software subsystem of the Mission Computer. The Software subsystem includes all the software components (represented as AADL threads) that 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 (SW.Impl) specifies how each component is connected.

To start, we have the initial version of the architecture, along with each component's AGREE contracts, as depicted in Figure 9. This figure corresponds to a simplified version of the SW.aadl model (the position status information pathway and peripheral devices have been omitted for clarity). You may view the graphical model of this initial implementation by opening the SW_SW_Impl.aadl_diagram file in the CASE_Simple_Example_V2/diagrams/ directory in the Navigation pane.

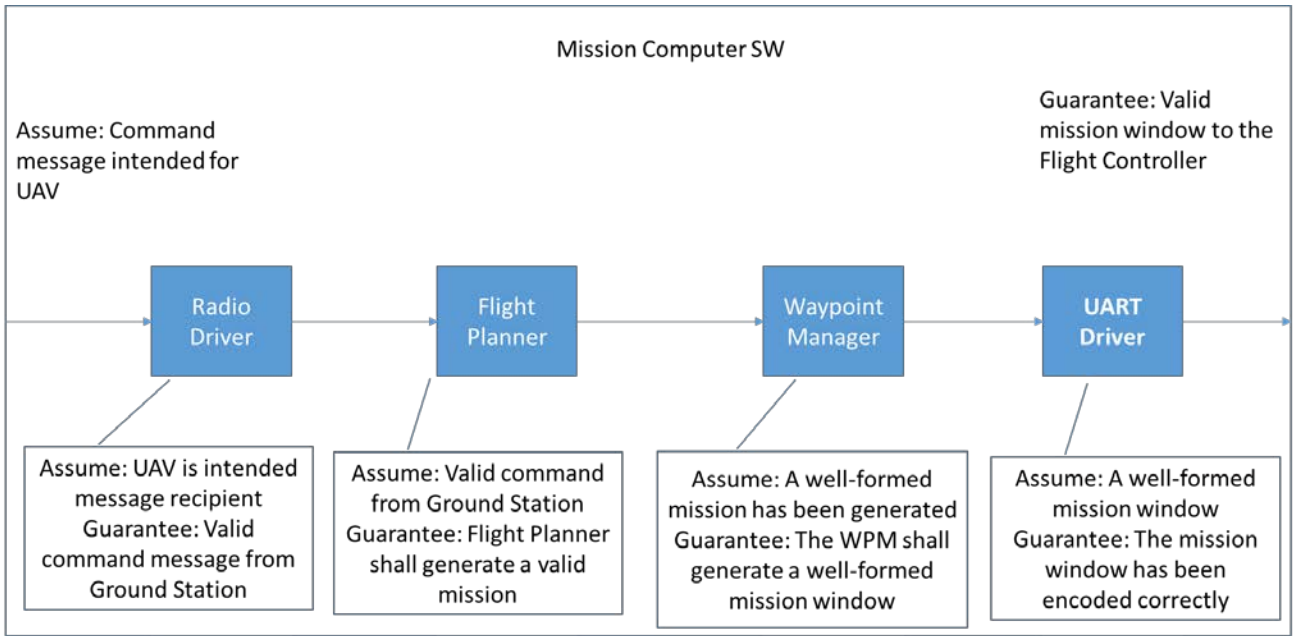
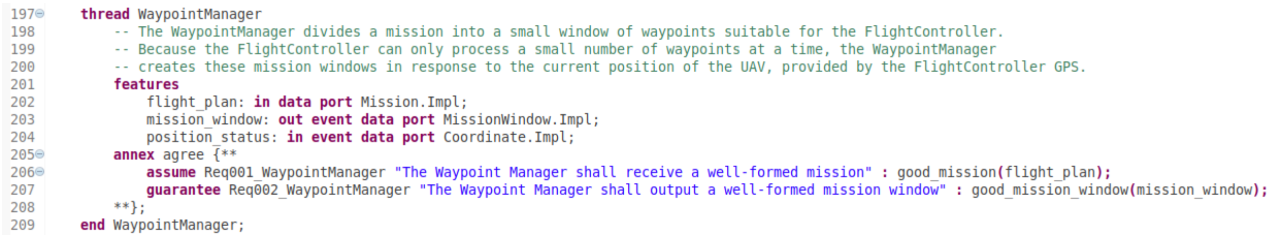

Mission Computer SW
Assume: Command message intended for UAV
Guarantee: Valid mission window to the Flight Controller
Radio Driver
Flight Planner
Waypoint Manager
UART Driver
Assume: UAV is intended message recipient
Guarantee: Valid command message from Ground Station
Assume: Valid command from Ground Station
Guarantee: Flight Planner shall generate a valid mission
Assume: A well-formed mission has been generated
Guarantee: The WPM shall generate a well-formed mission window
Assume: A well-formed mission window
Guarantee: The mission window has been encoded correctly

Figure 9. Initial SW Architecture.

If we look at one of the components, for example the Waypoint Manager, its specification (as shown in Figure 10) describes its data ports and associated message types, as well as the AGREE contract it must satisfy.



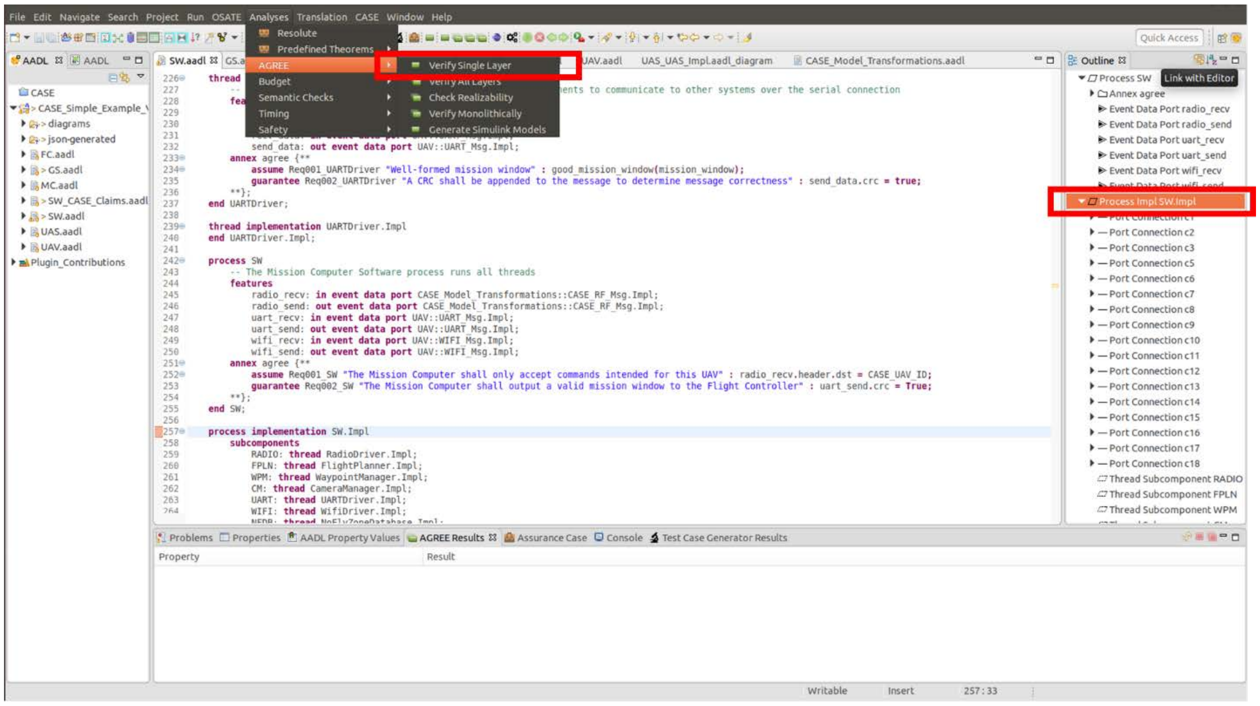
```
thread WaypointManager
    -- The WaypointManager divides a mission into a small window of waypoints suitable for the FlightController.
    -- Because the FlightController can only process a small number of waypoints at a time, the WaypointManager
    -- creates these mission windows in response to the current position of the UAV, provided by the FlightController GPS.
    features
        flight_plan: in data port Mission.Impl;
        mission_window: out event data port MissionWindow.Impl;
        position_status: in event data port Coordinate.Impl;
    annex agree {**
        assume Req001_WaypointManager "The Waypoint Manager shall receive a well-formed mission" : good_mission(flight_plan);
        guarantee Req002_WaypointManager "The Waypoint Manager shall output a well-formed mission window" : good_mission_window(mission_window);
    **};
end WaypointManager;
```

Figure 10. 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 as output for the Flight Controller.

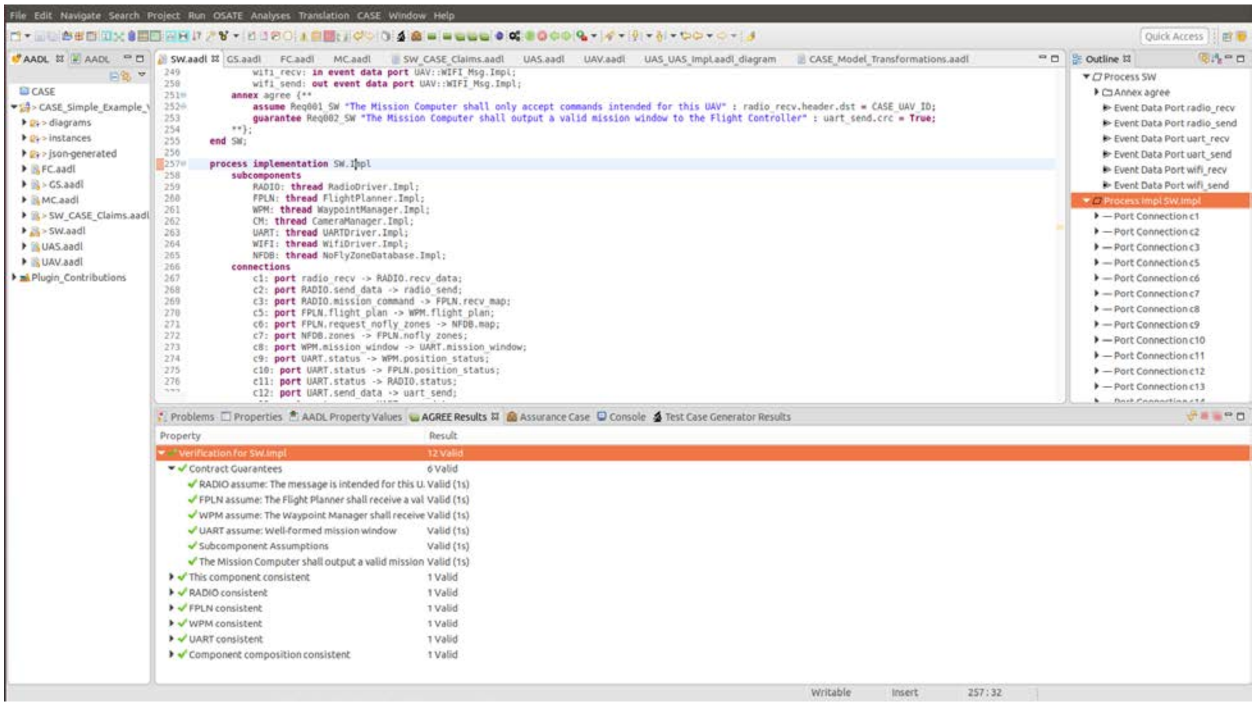
The top-level assumption and guarantee in the upper corners of Figure 9 state that the Mission Computer Software will guarantee the generation of a valid mission window, assuming it is the intended recipient of the command message received from the Ground Station. The guarantee is only possible due to 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 information pathway can guarantee it will produce a valid mission window. If a component cannot guarantee a property that the successive component on the information pathway assumes, AGREE will catch the property violation.

In order to demonstrate that our initial architecture satisfies the requirements listed in Table 1, we perform an AGREE analysis. To do this, select the "Process Impl SW.Impl" item in the Outline pane and then select Analyses → AGREE → Verify Single Layer from the OSATE menu (see Figure 11). The output will appear in the Output pane at the bottom of the window (Figure 12). Note that the green check marks next to each item indicate that the properties are valid (a red '!' will indicate failure).



```
thread
    --                                    ents to communicate to other systems over the serial connection
    fea

        send_data: out event data port UAV::UART_Msg.Impl;
    annex agree {**
        assume Req001_UARTDriver "Well-formed mission window" : good_mission_window(mission_window);
        guarantee Req002_UARTDriver "A CRC shall be appended to the message to determine message correctness" : send_data.crc = true;
    **};
end UARTDriver;

thread implementation UARTDriver.Impl
end UARTDriver.Impl;

process SW
    -- The Mission Computer Software process runs all threads
    features
        radio_recv: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
        radio_send: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
        uart_recv: in event data port UAV::UART_Msg.Impl;
        uart_send: out event data port UAV::UART_Msg.Impl;
        wifi_recv: in event data port UAV::WIFI_Msg.Impl;
        wifi_send: out event data port UAV::WIFI_Msg.Impl;
    annex agree {**
        assume Req001_SW "The Mission Computer shall only accept commands intended for this UAV" : radio_recv.header.dst = CASE_UAV_ID;
        guarantee Req002_SW "The Mission Computer shall output a valid mission window to the Flight Controller" : uart_send.crc = True;
    **};
end SW;

process implementation SW.Impl
    subcomponents
        RADIO: thread RadioDriver.Impl;
        FPLN: thread FlightPlanner.Impl;
        WPM: thread WaypointManager.Impl;
        CM: thread CameraManager.Impl;
        UART: thread UARTDriver.Impl;
        WIFI: thread WifiDriver.Impl;
        NFDB: thread NoFlyZoneDatabase.Impl;
```

Figure 11. Running AGREE.



```
            wifi_recv: in event data port UAV::WIFI_Msg.Impl;
            wifi_send: out event data port UAV::WIFI_Msg.Impl;
        annex agree {**
            assume Req001_SW "The Mission Computer shall only accept commands intended for this UAV" : radio_recv.header.dst = CASE_UAV_ID;
            guarantee Req002_SW "The Mission Computer shall output a valid mission window to the Flight Controller" : uart_send.crc = True;
        **};
    end SW;

    process implementation SW.Impl
        subcomponents
            RADIO: thread RadioDriver.Impl;
            FPLN: thread FlightPlanner.Impl;
            WPM: thread WaypointManager.Impl;
            CM: thread CameraManager.Impl;
            UART: thread UARTDriver.Impl;
            WIFI: thread WifiDriver.Impl;
            NFDB: thread NoFlyZoneDatabase.Impl;
        connections
            c1: port radio_recv -> RADIO.recv_data;
            c2: port RADIO.send_data -> radio_send;
            c3: port RADIO.mission_command -> FPLN.recv_map;
            c5: port FPLN.flight_plan -> WPM.flight_plan;
            c6: port FPLN.request_nofly_zones -> NFDB.map;
            c7: port NFDB.zones -> FPLN.nofly_zones;
            c8: port WPM.mission_window -> UART.mission_window;
            c9: port UART.status -> WPM.position_status;
            c10: port UART.status -> FPLN.position_status;
            c11: port UART.status -> RADIO.status;
            c12: port UART.send_data -> uart_send;
```

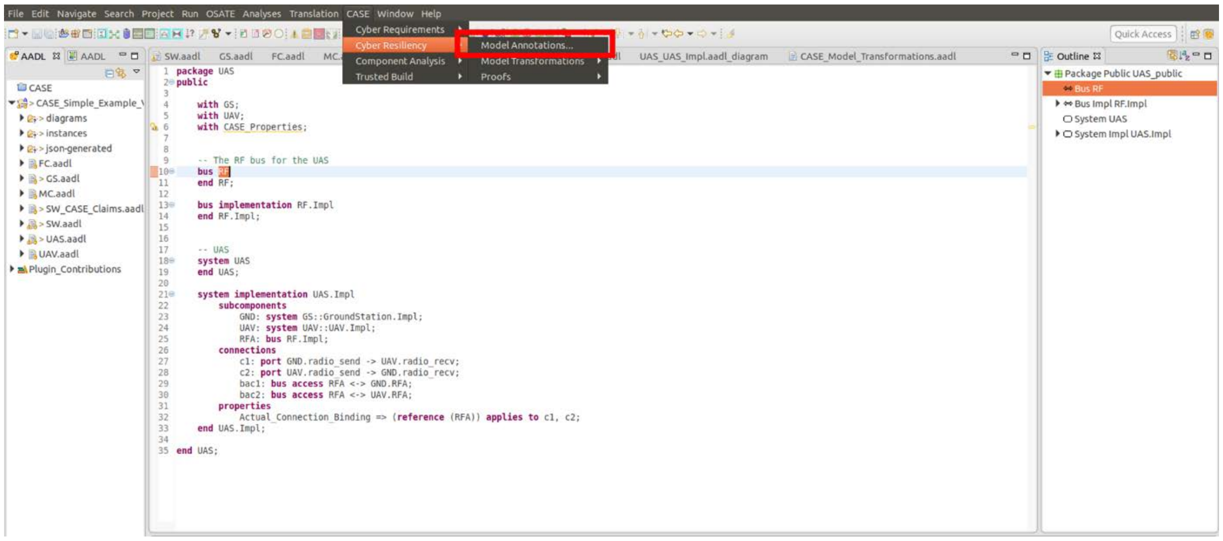
```
Property                                                   Result
Verification for SW.Impl                                   12 Valid
  Contract Guarantees                                      6 Valid
    RADIO assume: The message is intended for this U.      Valid (1s)
    FPLN assume: The Flight Planner shall receive a val    Valid (1s)
    WPM assume: The Waypoint Manager shall receive         Valid (1s)
    UART assume: Well-formed mission window                Valid (1s)
    Subcomponent Assumptions                               Valid (1s)
    The Mission Computer shall output a valid mission      Valid (1s)
  This component consistent                                1 Valid
  RADIO consistent                                         1 Valid
  FPLN consistent                                          1 Valid
  WPM consistent                                           1 Valid
  UART consistent                                          1 Valid
  Component composition consistent                         1 Valid
```

Figure 12. AGREE output.

Now that we are convinced that our current design satisfies its requirements, we wish to perform an analysis to look for requirement gaps, specifically with respect to cyber-security. Our development environment enables the addition of model analysis tools, and we can run our design through a TA1 tool, such as Charles River Analytics' GearCASE tool.

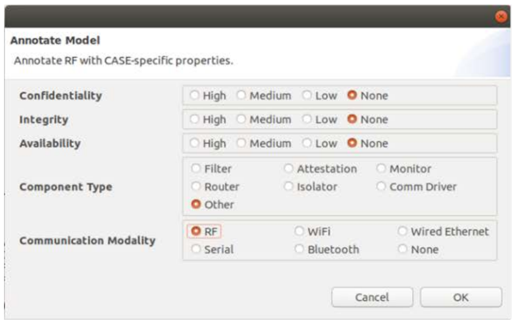
In discussions with TA1 performers, we have identified scenarios in which it will be useful to provide additional details about our system. Failing to do so could result in the generation of up to hundreds of additional cyber requirements that ultimately will be redundant or irrelevant, and will consume valuable engineering hours to address. Although we are still in the process of working with our TA1 and TA6 partners to determine what kinds of annotations are reasonable, we have developed an initial interface for easily and annotating the AADL model with properties for aiding cyber analysis.

In our example, we would like to indicate that the communication channel that the Ground Station uses to communicate with the UAV is radio. The TA1 tool may use this information to generate requirements specific to RF channels, while omitting others that may only apply to other communication modalities. We therefore wish to annotate the model to specify RF. Open the UAS.aadl file. This is the top-level AADL model that contains the entire system, including the Ground Station and UAV, as well as the communication bus that connects them. In the text editor, or Outline pane, select the Bus RF classifier, as illustrated in Figure 13. Next, click on the CASE → Cyber Resiliency → Model Annotations… menu, which will bring up the form shown in Figure 14a. For Communication Modality, select the RF option and click OK. The RF bus should now include the COMM_MODALITY => RF property (see Figure 14b).



```
package UAS
public

    with GS;
    with UAV;
    with CASE_Properties;


    -- The RF bus for the UAS
    bus RF
    end RF;

    bus implementation RF.Impl
    end RF.Impl;


    -- UAS
    system UAS
    end UAS;

    system implementation UAS.Impl
        subcomponents
            GND: system GS::GroundStation.Impl;
            UAV: system UAV::UAV.Impl;
            RFA: bus RF.Impl;
        connections
            c1: port GND.radio_send -> UAV.radio_recv;
            c2: port UAV.radio_send -> GND.radio_recv;
            bac1: bus access RFA <-> GND.RFA;
            bac2: bus access RFA <-> UAV.RFA;
        properties
            Actual_Connection_Binding => (reference (RFA)) applies to c1, c2;
    end UAS.Impl;

end UAS;
```

Figure 13. Adding CASE Model Annotations

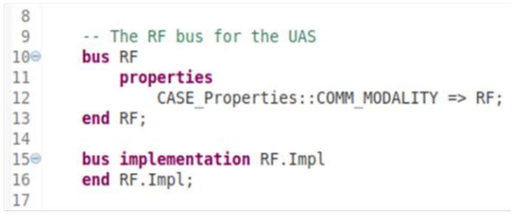


Annotate Model

Annotate RF with CASE-specific properties.

Confidentiality	High Medium Low [✓ None]
Integrity	High Medium Low [✓ None]
Availability	High Medium Low [✓ None]
Component Type	Filter Attestation Monitor Router Isolator Comm Driver [✓ Other]
Communication Modality	[✓ RF] WiFi Wired Ethernet Serial Bluetooth None

(a)

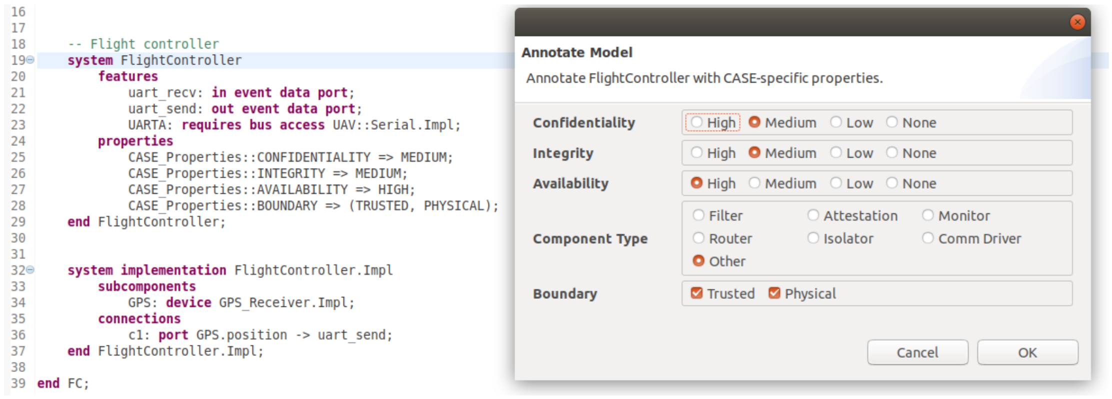


```
    -- The RF bus for the UAS
    bus RF
        properties
            CASE_Properties::COMM_MODALITY => RF;
    end RF;

    bus implementation RF.Impl
    end RF.Impl;
```

(b)

Figure 14. (a) CASE model annotation wizard. (b) Annotated model.

Different annotations will be available for insertion depending on the component selected. The FlightController, for example has *physical* and *trusted* boundaries specified, as well as values for *confidentiality*, *integrity*, and *availability*, as shown in Figure 15.



```


    -- Flight controller
    system FlightController
        features
            uart_recv: in event data port;
            uart_send: out event data port;
            UARTA: requires bus access UAV::Serial.Impl;
        properties
            CASE_Properties::CONFIDENTIALITY => MEDIUM;
            CASE_Properties::INTEGRITY => MEDIUM;
            CASE_Properties::AVAILABILITY => HIGH;
            CASE_Properties::BOUNDARY => (TRUSTED, PHYSICAL);
    end FlightController;


    system implementation FlightController.Impl
        subcomponents
            GPS: device GPS_Receiver.Impl;
        connections
            c1: port GPS.position -> uart_send;
    end FlightController.Impl;

end FC;
```

Annotate Model

Annotate FlightController with CASE-specific properties.

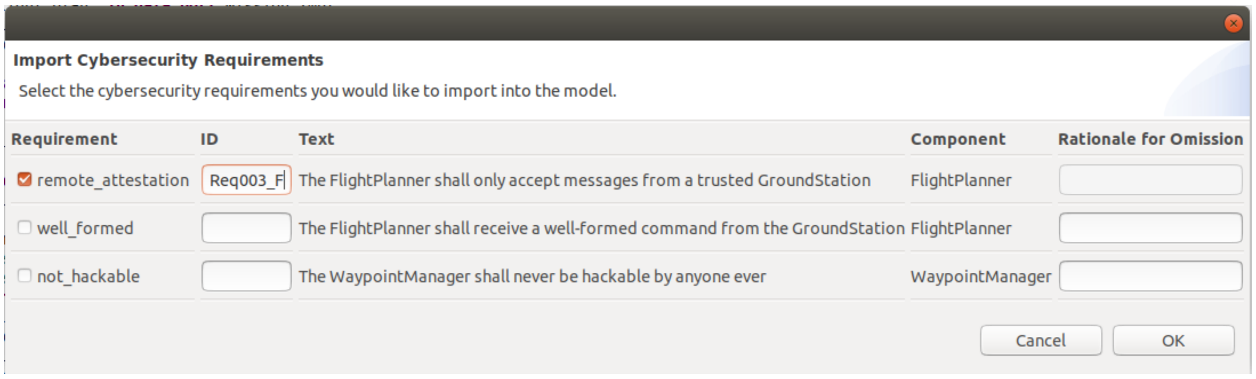
Confidentiality	High [✓ Medium] Low None
Integrity	High [✓ Medium] Low None
Availability	[✓ High] Medium Low None
Component Type	Filter Attestation Monitor Router Isolator Comm Driver [✓ Other]
Boundary	[✓ Trusted] [✓ Physical]

Figure 15. FlightController annotations.

Return to the SW.aadl file before proceeding.

Although Requirements tools such as GearCASE are not yet integrated with our environment, we have created the interface for accepting TA1 tool output. For purposes of this demo, we've hard-coded three requirements, which can be viewed by selecting CASE → Cyber Requirements → Run TA1 tool from the OSATE menu. The TA1 tool will analyze the model, identify gaps in cyber requirements, and present them to the developer. Because some of the generated requirements will not be applicable to the system under development, the developer is provided with an option to import each requirement into the model, or to ignore the requirement. If the user chooses to ignore a requirement, rationale can be entered and saved to a log file for traceability (requirement omission log generation is not yet implemented).

Figure 16 shows the current version of the cyber requirement input form. Each requirement includes a check box to indicate whether to import the requirement, a short name, a text description, the applicable model component, and a text box to input omission rationale. Future versions of this form will organize the requirements in a manner that minimizes the burden of iterating through potentially hundreds of new requirements. For purposes of this demo, three missing cyber requirements were identified. The first specifies that the Flight Planner shall only receive messages from trusted sources. The second specifies that the Flight Planner shall only receive well-formed messages. We agree that these are desired properties for our architecture, and indicate that we wish to address them by checking the boxes. The third requirement is too broad to be practical, so we choose to ignore it by leaving the box unchecked and providing rationale. Note that rationale can only be entered for requirements that are not selected. Although the requirement import tool can import multiple requirements at once, for purposes of this demo we will import them one at a time in order to better showcase the model transformations that address them. Make sure only the *remote_attestation* requirement is selected, and a requirement ID is entered (e.g., Req003_FlightPlanner), then click the OK button and the form will close.


Import Cybersecurity Requirements

Select the cybersecurity requirements you would like to import into the model.

Requirement	ID	Text	Component	Rationale for Omission
✓ remote_attestation	Req003_F	The FlightPlanner shall only accept messages from a trusted GroundStation	FlightPlanner	
☐ well_formed		The FlightPlanner shall receive a well-formed command from the GroundStation	FlightPlanner	
☐ not_hackable		The WaypointManager shall never be hackable by anyone ever	WaypointManager	

Figure 16. CASE cybersecurity requirements import form

The Flight Planner component in the model is now modified in two places, as shown in Figure 17. First, an AGREE *assume* statement has been added, that includes a textual description of the requirement:

"The FlightPlanner shall only accept messages from a trusted GroundStation."

The assume statement should also include a formalized version of the requirement, however, the definition of *trusted* cannot be determined by the requirements tool and must be manually entered. In its place, the tool instead inserts the FALSE keyword. If we were to run AGREE with the current assume statement, it would fail. This is desired, because it forces the developer to address the incomplete requirement.

The second modification involves the mechanism for both enforcing and assessing whether the proper development activities have been performed: the assurance case. For each imported requirement, a Resolute clause is inserted in the corresponding component type. At the end of (or during) development, the Resolute tool can generate an assurance argument that the requirement was properly addressed. You can see the generated claim for the requirement by either clicking on the *remote_attestation* function name in the prove statement and pressing the F3 key, or by opening the SW_CASE_Claims.aadl file.

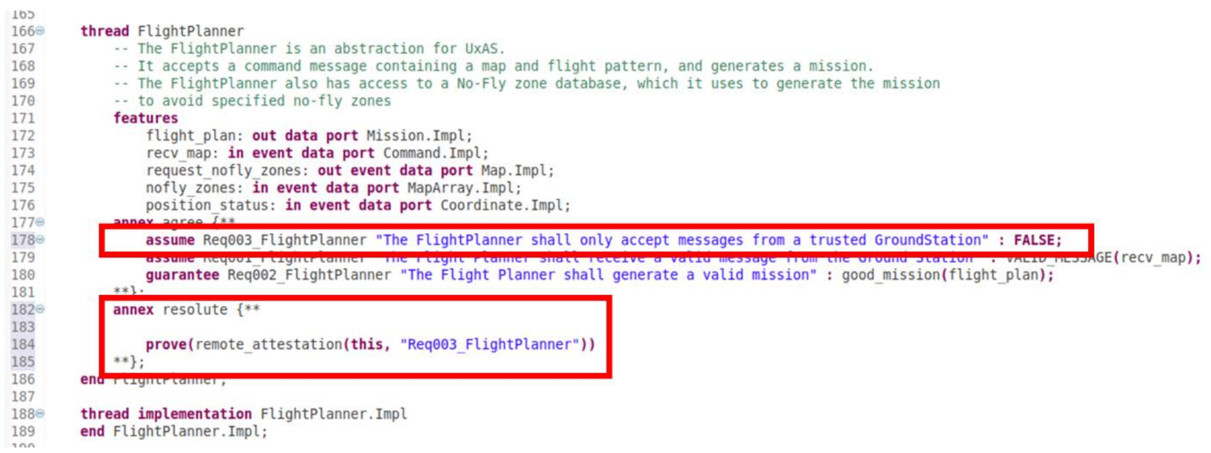
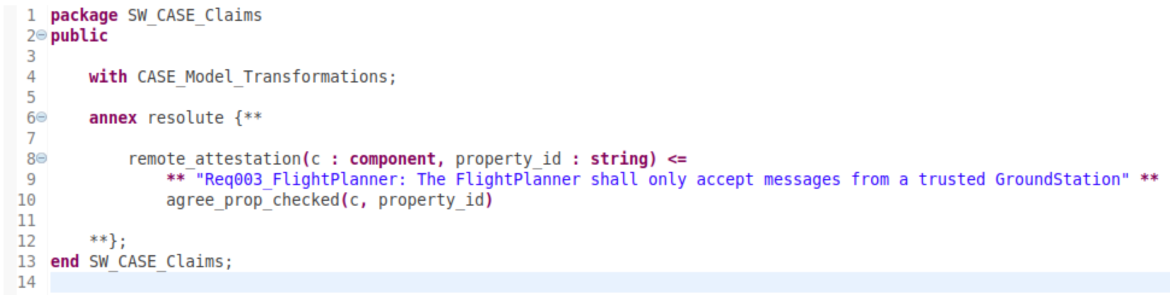

```
thread FlightPlanner
    -- The FlightPlanner is an abstraction for UxAS.
    -- It accepts a command message containing a map and flight pattern, and generates a mission.
    -- The FlightPlanner also has access to a No-Fly zone database, which it uses to generate the mission
    -- to avoid specified no-fly zones
    features
        flight_plan: out data port Mission.Impl;
        recv_map: in event data port Command.Impl;
        request_nofly_zones: out event data port Map.Impl;
        nofly_zones: in event data port MapArray.Impl;
        position_status: in event data port Coordinate.Impl;
    annex agree {**
        assume Req003_FlightPlanner "The FlightPlanner shall only accept messages from a trusted GroundStation" : FALSE;
        assume Req001_FlightPlanner "The Flight Planner shall receive a valid message from the Ground Station" : VALID_MESSAGE(recv_map);
        guarantee Req002_FlightPlanner "The Flight Planner shall generate a valid mission" : good_mission(flight_plan);
    **};
    annex resolute {**

        prove(remote_attestation(this, "Req003_FlightPlanner"))
    **};
end FlightPlanner;

thread implementation FlightPlanner.Impl
end FlightPlanner.Impl;
```

Figure 17. Importing a cybersecurity requirement into the model

At first, the assurance argument is mostly bare. It only includes a statement that the corresponding AGREE property must pass. However, as we address the requirement by performing model transformations and other development, test, and build activities, the claim will grow more complex. Figure 18 shows the remote_attestation claim.



```
package SW_CASE_Claims
public

    with CASE_Model_Transformations;

    annex resolute {**

        remote_attestation(c : component, property_id : string) <=
            ** "Req003_FlightPlanner: The FlightPlanner shall only accept messages from a trusted GroundStation" **
            agree_prop_checked(c, property_id)

    **};
end SW_CASE_Claims;
```

Figure 18. Resolute claim for remote_attestation.

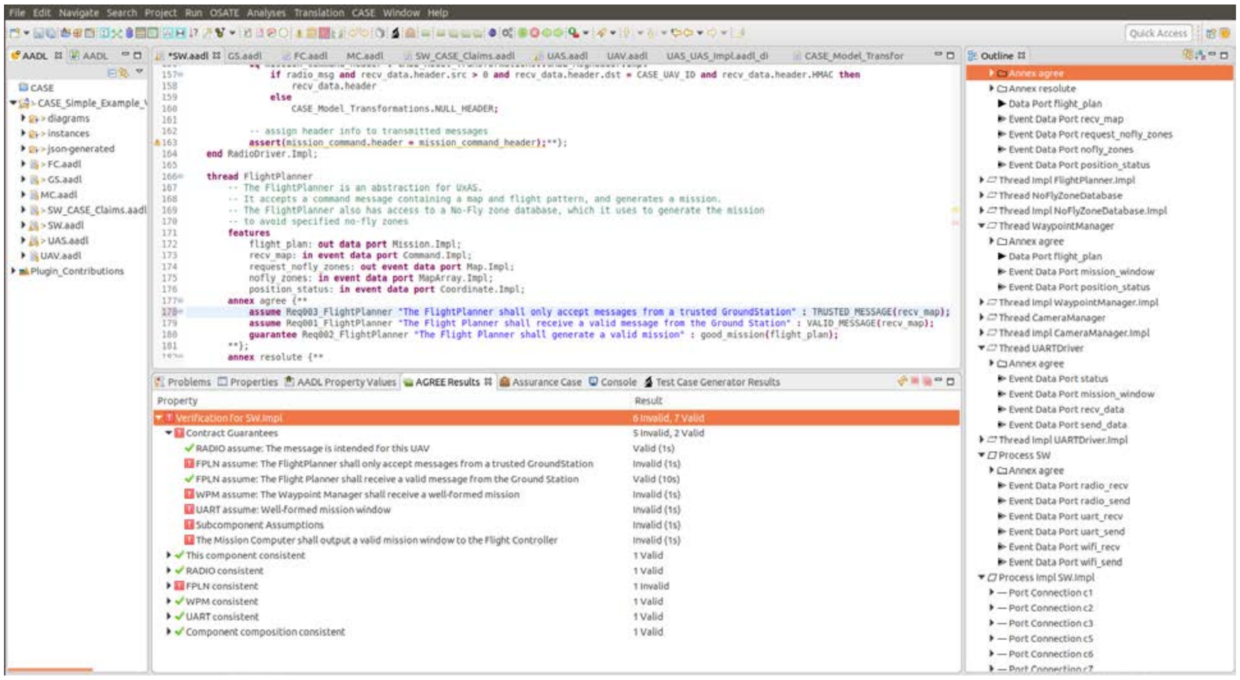
Recall that this new requirement specifies that the Flight Planner shall only receive trusted messages from the Ground Station. To formalize this into the model, we must complete the new AGREE assume statement. Replace the FALSE keyword with the formal property so that the assume statement appears as below:

```
assume Req003_FlightPlanner "The FlightPlanner shall only accept messages from a trusted GroundStation" : TRUSTED_MESSAGE(recv_map);
```

The definition of TRUSTED_MESSAGE() needs to be provided by the developer. For the demo, it has been included in the AGREE library annex at the bottom of SW.aadl.

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 valid *and* trusted. For a message to be trusted, the Ground Station must pass remote attestation.

Before modifying the design to address remote attestation, save the model and run AGREE again (remember that the "Process Impl MC_SW.Impl" item in the Outline pane must be selected. You should observe the results shown in Figure 19. Note that the red exclamation marks indicate that the property failed.



```
                if radio_msg and recv_data.header.src > 0 and recv_data.header.dst = CASE_UAV_ID and recv_data.header.HMAC then
                    recv_data.header
                else
                    CASE_Model_Transformations.NULL_HEADER;

            -- assign header info to transmitted messages
            assert(mission_command.header = mission_command_header);**};
    end RadioDriver.Impl;

    thread FlightPlanner
        -- The FlightPlanner is an abstraction for UxAS.
        -- It accepts a command message containing a map and flight pattern, and generates a mission.
        -- The FlightPlanner also has access to a No-Fly zone database, which it uses to generate the mission
        -- to avoid specified no-fly zones
        features
            flight_plan: out data port Mission.Impl;
            recv_map: in event data port Command.Impl;
            request_nofly_zones: out event data port Map.Impl;
            nofly_zones: in event data port MapArray.Impl;
            position_status: in event data port Coordinate.Impl;
        annex agree {**
            assume Req003_FlightPlanner "The FlightPlanner shall only accept messages from a trusted GroundStation" : TRUSTED_MESSAGE(recv_map);
            assume Req001_FlightPlanner "The Flight Planner shall receive a valid message from the Ground Station" : VALID_MESSAGE(recv_map);
            guarantee Req002_FlightPlanner "The Flight Planner shall generate a valid mission" : good_mission(flight_plan);
        **};
        annex resolute {**
```

```
Property                                                                                   Result
Verification for SW.Impl                                                                   6 Invalid, 7 Valid
  Contract Guarantees                                                                      5 Invalid, 2 Valid
    ✓ RADIO assume: The message is intended for this UAV                                   Valid (1s)
    FPLN assume: The FlightPlanner shall only accept messages from a trusted GroundStation  Invalid (1s)
    ✓ FPLN assume: The Flight Planner shall receive a valid message from the Ground Station Valid (10s)
    WPM assume: The Waypoint Manager shall receive a well-formed mission                   Invalid (1s)
    UART assume: Well-formed mission window                                                Invalid (1s)
    Subcomponent Assumptions                                                               Invalid (1s)
    The Mission Computer shall output a valid mission window to the Flight Controller      Invalid (1s)
  ✓ This component consistent                                                              1 Valid
  ✓ RADIO consistent                                                                       1 Valid
  FPLN consistent                                                                          1 Invalid
  ✓ WPM consistent                                                                         1 Valid
  ✓ UART consistent                                                                        1 Valid
  ✓ Component composition consistent                                                       1 Valid
```

Figure 19. Failed property in AGREE.

The failure occurred because we have no component in our architecture that will block messages from untrusted sources. To address this, we must add an *attestation manager* component that ensures only command messages from trusted sources reach the Flight Planner, as depicted in Figure 20. The attestation manager component will include the AGREE guarantee necessary to satisfy the assumption specified by the Flight Planner.

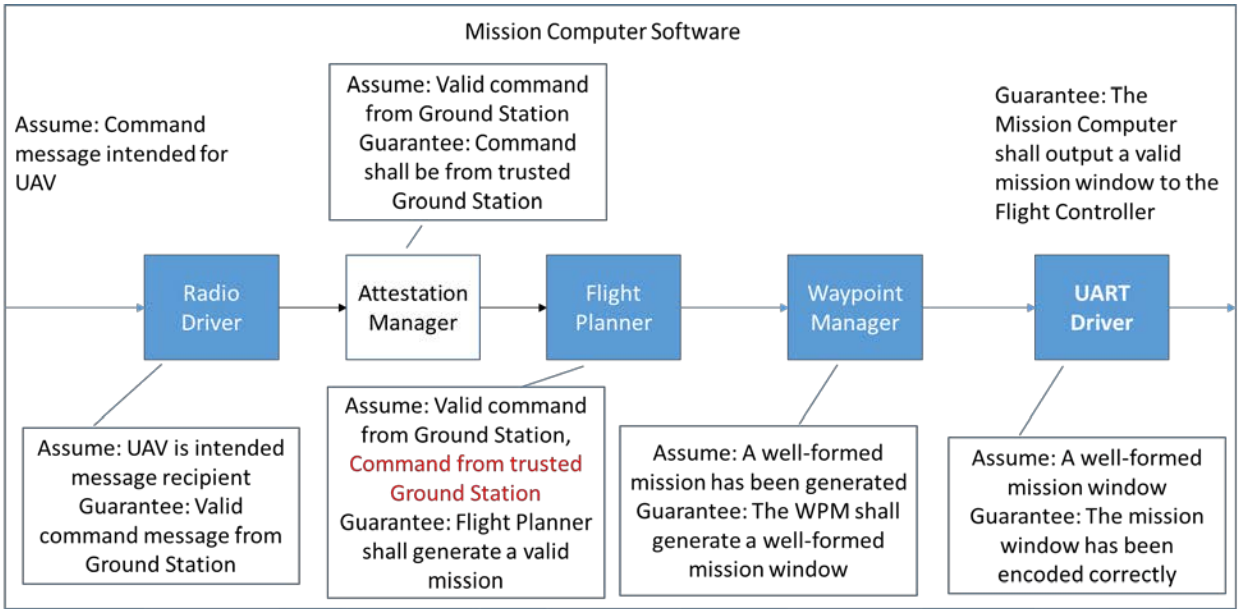
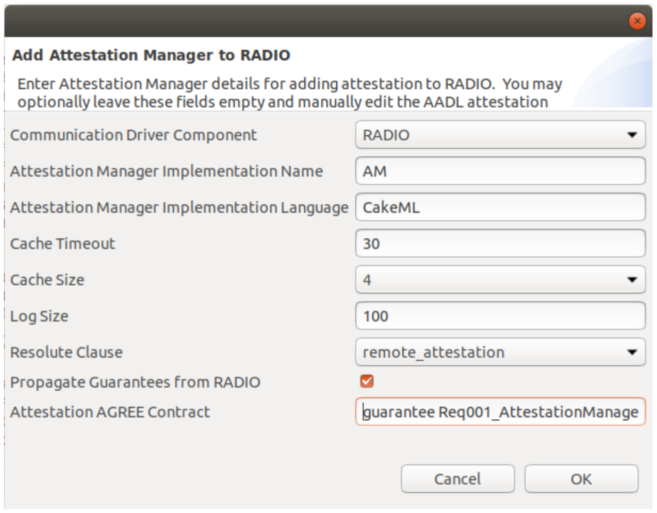

Mission Computer Software
Assume: Command message intended for UAV
Assume: Valid command from Ground Station Guarantee: Command shall be from trusted Ground Station
Guarantee: The Mission Computer shall output a valid mission window to the Flight Controller
Radio Driver
Attestation Manager
Flight Planner
Waypoint Manager
UART Driver
Assume: UAV is intended message recipient Guarantee: Valid command message from Ground Station
Assume: Valid command from Ground Station, Command from trusted Ground Station Guarantee: Flight Planner shall generate a valid mission
Assume: A well-formed mission has been generated Guarantee: The WPM shall generate a well-formed mission window
Assume: A well-formed mission window Guarantee: The mission window has been encoded correctly

Figure 20. Revised Architecture.

Although the Attestation Manager is added to satisfy the Flight Planner assumption, it is really bound to a communication driver, in this case the RadioDriver component. The Attestation Manager guarantees that *any* external message received by the radio hardware and processed in software through the Radio Driver will only be consumed if the message sender is trusted. The Attestation Manager works by maintaining a cache of records containing a message sender ID, the attestation result, and a timeout value, after which the attestation result in invalid. Upon receiving a message from a source that is not in the cache, the Attestation Manager sends an *attestation request* back to the message source, in our case, a Ground Station. The Ground Station provides an attestation response in a form that enables the attestation manager to assess the trustworthiness of the Ground Station. The Attestation Manager then records the Ground Station ID and attestation status in the cache and subsequent messages are either forwarded on to the Flight Planner, or dropped. To perform the Attestation model transformation, select the RADIO subcomponent in the SW.Impl implementation, and click the CASE → Cyber Resiliency → Model Transformations → Add Attestation Manager… menu item. A wizard will open, as shown in Figure 21.


Add Attestation Manager to RADIO

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

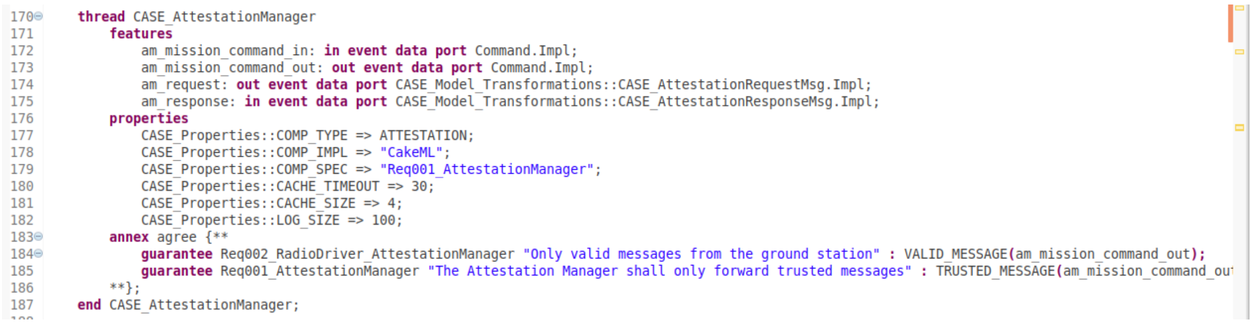
Communication Driver Component	RADIO
Attestation Manager Implementation Name	AM
Attestation Manager Implementation Language	CakeML
Cache Timeout	30
Cache Size	4
Log Size	100
Resolute Clause	remote_attestation
Propagate Guarantees from RADIO	✓
Attestation AGREE Contract	guarantee Req001_AttestationManage

Figure 21. CASE Attestation Manager wizard.

Fill in the form so that it appears as in Figure 21, and click OK. For the Attestation AGREE Contract, enter the following Guarantee statement (without line breaks):

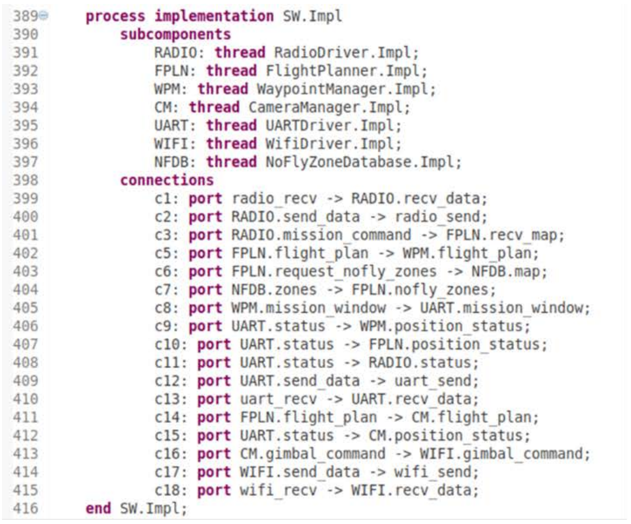
```
guarantee Req001_AttestationManager "The Attestation Manager shall only forward trusted messages"
: TRUSTED_MESSAGE(am_mission_command_out) and
(CASE_Model_Transformations.TRUSTED(am_mission_command_out.header.src) =>
(am_mission_command_out = am_mission_command_in));
```

Save the model. The SW.aadl model will now be revised to include a CASE_AttestationManager component type and implementation. Figure 22 displays the Attestation Manager component type declaration. To better distinguish between the previous implementation (without the attestation manager), an entirely new SW implementation was generated. Figure 23 shows the original implementation side-by-side with the transformed implementation. Note the additional “AM” subcomponent that has been added to SW.Impl1, as well as additional connections wiring the Attestation Manager to the Radio Driver and Flight Planner. You may view the graphical model of this implementation by opening the SW_SW_Impl1.aadl_diagram file in the CASE_Simple_Example_V2/diagrams/ directory in the Navigation pane.



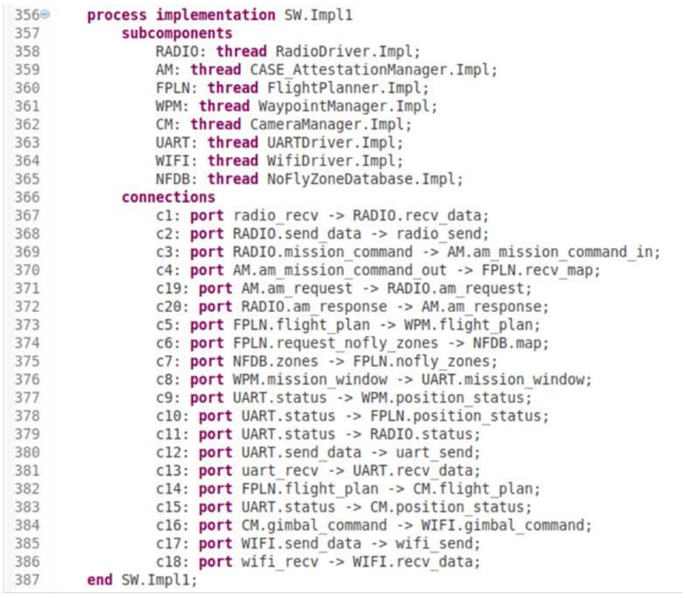
```
thread CASE_AttestationManager
    features
        am_mission_command_in: in event data port Command.Impl;
        am_mission_command_out: out event data port Command.Impl;
        am_request: out event data port CASE_Model_Transformations::CASE_AttestationRequestMsg.Impl;
        am_response: in event data port CASE_Model_Transformations::CASE_AttestationResponseMsg.Impl;
    properties
        CASE_Properties::COMP_TYPE => ATTESTATION;
        CASE_Properties::COMP_IMPL => "CakeML";
        CASE_Properties::COMP_SPEC => "Req001_AttestationManager";
        CASE_Properties::CACHE_TIMEOUT => 30;
        CASE_Properties::CACHE_SIZE => 4;
        CASE_Properties::LOG_SIZE => 100;
    annex agree {**
        guarantee Req002_RadioDriver_AttestationManager "Only valid messages from the ground station" : VALID_MESSAGE(am_mission_command_out);
        guarantee Req001_AttestationManager "The Attestation Manager shall only forward trusted messages" : TRUSTED_MESSAGE(am_mission_command_ou
    **};
end CASE_AttestationManager;
```

Figure 22. Attestation Manager component type declaration.



```
process implementation SW.Impl
    subcomponents
        RADIO: thread RadioDriver.Impl;
        FPLN: thread FlightPlanner.Impl;
        WPM: thread WaypointManager.Impl;
        CM: thread CameraManager.Impl;
        UART: thread UARTDriver.Impl;
        WIFI: thread WifiDriver.Impl;
        NFDB: thread NoFlyZoneDatabase.Impl;
    connections
        c1: port radio_recv -> RADIO.recv_data;
        c2: port RADIO.send_data -> radio_send;
        c3: port RADIO.mission_command -> FPLN.recv_map;
        c5: port FPLN.flight_plan -> WPM.flight_plan;
        c6: port FPLN.request_nofly_zones -> NFDB.map;
        c7: port NFDB.zones -> FPLN.nofly_zones;
        c8: port WPM.mission_window -> UART.mission_window;
        c9: port UART.status -> WPM.position_status;
        c10: port UART.status -> FPLN.position_status;
        c11: port UART.status -> RADIO.status;
        c12: port UART.send_data -> uart_send;
        c13: port uart_recv -> UART.recv_data;
        c14: port FPLN.flight_plan -> CM.flight_plan;
        c15: port UART.status -> CM.position_status;
        c16: port CM.gimbal_command -> WIFI.gimbal_command;
        c17: port WIFI.send_data -> wifi_send;
        c18: port wifi_recv -> WIFI.recv_data;
end SW.Impl;
```

(a)



```
process implementation SW.Impl1
    subcomponents
        RADIO: thread RadioDriver.Impl;
        AM: thread CASE_AttestationManager.Impl;
        FPLN: thread FlightPlanner.Impl;
        WPM: thread WaypointManager.Impl;
        CM: thread CameraManager.Impl;
        UART: thread UARTDriver.Impl;
        WIFI: thread WifiDriver.Impl;
        NFDB: thread NoFlyZoneDatabase.Impl;
    connections
        c1: port radio_recv -> RADIO.recv_data;
        c2: port RADIO.send_data -> radio_send;
        c3: port RADIO.mission_command -> AM.am_mission_command_in;
        c4: port AM.am_mission_command_out -> FPLN.recv_map;
        c19: port AM.am_request -> RADIO.am_request;
        c20: port RADIO.am_response -> AM.am_response;
        c5: port FPLN.flight_plan -> WPM.flight_plan;
        c6: port FPLN.request_nofly_zones -> NFDB.map;
        c7: port NFDB.zones -> FPLN.nofly_zones;
        c8: port WPM.mission_window -> UART.mission_window;
        c9: port UART.status -> WPM.position_status;
        c10: port UART.status -> FPLN.position_status;
        c11: port UART.status -> RADIO.status;
        c12: port UART.send_data -> uart_send;
        c13: port uart_recv -> UART.recv_data;
        c14: port FPLN.flight_plan -> CM.flight_plan;
        c15: port UART.status -> CM.position_status;
        c16: port CM.gimbal_command -> WIFI.gimbal_command;
        c17: port WIFI.send_data -> wifi_send;
        c18: port wifi_recv -> WIFI.recv_data;
end SW.Impl1;
```

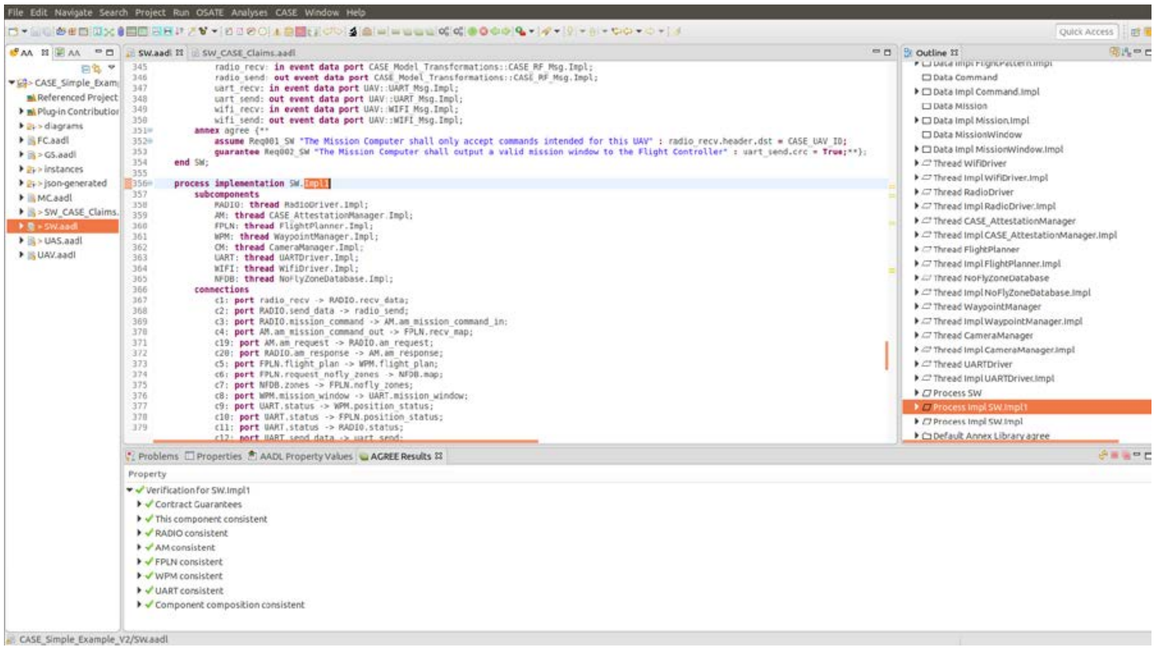
(b)

Figure 23. Software implementations. (a) Initial software. (b) Software including Attestation Manager.

You will notice that in addition to inserting the attestation manager on connection c3 in Figure 23a, two new connections for handling attestation requests and responses were added between the Radio Driver and Attestation Manager.

The inserted CASE_AttestationManager.Impl thread implementation contains a large AGREE annex that specifies the behavior of the attestation manager. This enables AGREE to analyze whether the component contracts are valid. We can now run AGREE by selecting Process Impl SW.Impl1 in the Outline pane, and clicking the Analyses → AGREE → Verify Single Layer menu item. The AGREE analysis should pass, and the results should appear as in Figure 24.

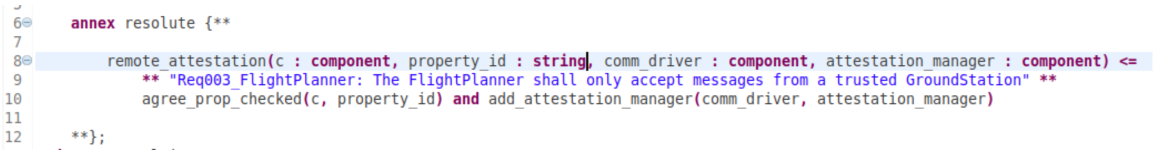
However, just because our architecture satisfies its assume-guarantee contracts, does not mean that we have adequately addressed the TA1 requirement. For example, it could be possible that some time after the attestation manager is added to the radio driver, new ports are added to the radio driver with connections that completely bypass the attestation manager. In order for the attestation manager model transformation to be effective, we need assurance at any time throughout development that it is positioned correctly. If you open the SW_CASE_Claims.aadl file again, you will notice the remote_attestation claim has changed. Now that we know we are addressing the requirement via the inclusion of an attestation manager, we can build an assurance case that it was done properly. An add_attestation_manager() function call was added to the claim, as can be seen in Figure 25. The function is defined in the CASE_Model_Transformations.aadl Plug-in Contribution, which can be opened from the Navigation pane, or by clicking on the add_attestation_manager() function name and pressing the F3 key.



```
            radio_recv: in event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
            radio_send: out event data port CASE_Model_Transformations::CASE_RF_Msg.Impl;
            uart_recv: in event data port UAV::UART_Msg.Impl;
            uart_send: out event data port UAV::UART_Msg.Impl;
            wifi_recv: in event data port UAV::WIFI_Msg.Impl;
            wifi_send: out event data port UAV::WIFI_Msg.Impl;
        annex agree {**
            assume Req001_SW "The Mission Computer shall only accept commands intended for this UAV" : radio_recv.header.dst = CASE_UAV_ID;
            guarantee Req002_SW "The Mission Computer shall output a valid mission window to the Flight Controller" : uart_send.crc = True;**};
    end SW;

    process implementation SW.Impl1
        subcomponents
            RADIO: thread RadioDriver.Impl;
            AM: thread CASE_AttestationManager.Impl;
            FPLN: thread FlightPlanner.Impl;
            WPM: thread WaypointManager.Impl;
            CM: thread CameraManager.Impl;
            UART: thread UARTDriver.Impl;
            WIFI: thread WifiDriver.Impl;
            NFDB: thread NoFlyZoneDatabase.Impl;
        connections
            c1: port radio_recv -> RADIO.recv_data;
            c2: port RADIO.send_data -> radio_send;
            c3: port RADIO.mission_command -> AM.am_mission_command_in;
            c4: port AM.am_mission_command_out -> FPLN.recv_map;
            c19: port AM.am_request -> RADIO.am_request;
            c20: port RADIO.am_response -> AM.am_response;
            c5: port FPLN.flight_plan -> WPM.flight_plan;
            c6: port FPLN.request_nofly_zones -> NFDB.map;
            c7: port NFDB.zones -> FPLN.nofly_zones;
            c8: port WPM.mission_window -> UART.mission_window;
            c9: port UART.status -> WPM.position_status;
            c10: port UART.status -> FPLN.position_status;
            c11: port UART.status -> RADIO.status;
            c12: port UART.send_data -> uart_send;
```

Property
- ✓ Verification for SW.Impl1
  - ✓ Contract Guarantees
  - ✓ This component consistent
  - ✓ RADIO consistent
  - ✓ AM consistent
  - ✓ FPLN consistent
  - ✓ WPM consistent
  - ✓ UART consistent
  - ✓ Component composition consistent

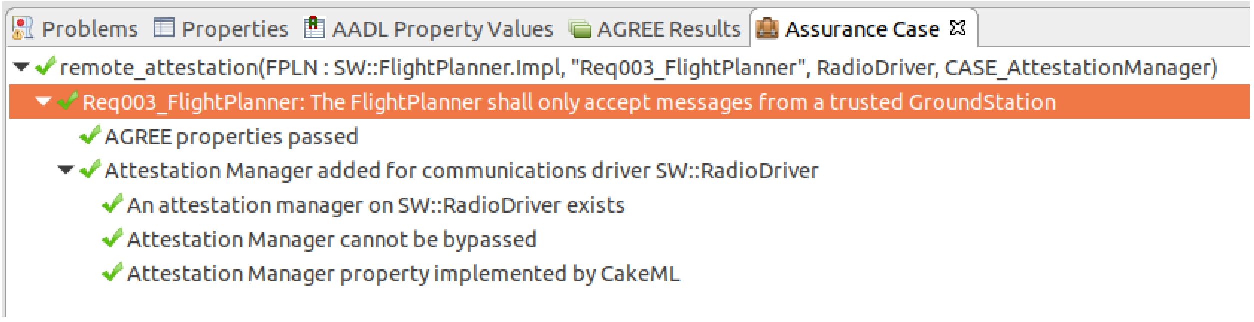
Figure 24. Passing AGREE analysis for model with Attestation Manager.



```
    annex resolute {**

        remote_attestation(c : component, property_id : string, comm_driver : component, attestation_manager : component) <=
            ** "Req003_FlightPlanner: The FlightPlanner shall only accept messages from a trusted GroundStation" **
            agree_prop_checked(c, property_id) and add_attestation_manager(comm_driver, attestation_manager)

    **};
```

Figure 25. Modified remote_attestation claim.

We can run Resolute by returning to the SW.aadl file, selecting the Process Impl SW.Impl1 in the Outline pane, and clicking the Analyses → Resolute menu item. From Figure 25, we see that Resolute will check two claims. The first is that the AGREE property (that was generated when the requirement was imported from the TA1 tool) is checked by AGREE and passes. The second is that the Attestation Manager was added correctly. By looking at the definition of add_attestation_manager(), we see three sub-claims: (1) an Attestation Manager exists in the model and is bound to the Radio Driver, (2) communications from the Radio Driver cannot bypass the Attestation Manager, and (3) the Attestation Manager must be implemented in CakeML. If any of these claims are untrue, Resolute will fail. We have designed a small plug-in call AgreeCheck that Resolute uses to not only determine that AGREE was run and the properties passed, but also that AGREE was run on the latest version of the model. Therefore, if a model is analyzed by AGREE and passes, but is then later modified, Resolute will fail until the model is re-run through AGREE to verify the modifications have no impact. Assuming the model was not modified since last running AGREE, the Resolute output should appear as in Figure 26. If the *AGREE properties passed* claim failed, re-run AGREE, then re-run Resolute.

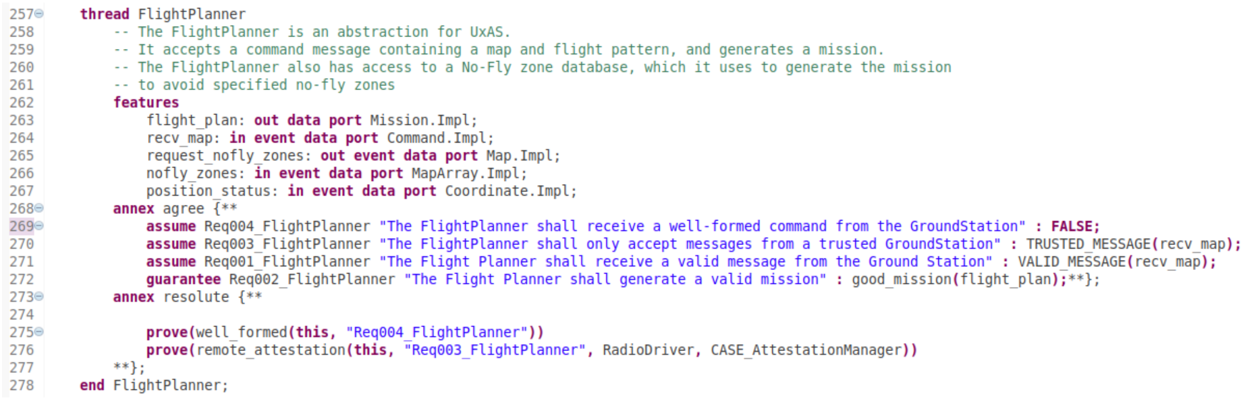


```
▼ ✓ remote_attestation(FPLN : SW::FlightPlanner.Impl, "Req003_FlightPlanner", RadioDriver, CASE_AttestationManager)
  ▼ ✓ Req003_FlightPlanner: The FlightPlanner shall only accept messages from a trusted GroundStation
      ✓ AGREE properties passed
    ▼ ✓ Attestation Manager added for communications driver SW::RadioDriver
        ✓ An attestation manager on SW::RadioDriver exists
        ✓ Attestation Manager cannot be bypassed
        ✓ Attestation Manager property implemented by CakeML
```

Figure 26. Resolute results.

We may now run our architecture through another iteration of analysis using TA1 tools (or import the other requirements from the initial analysis). By launching the TA1 tool from the CASE → Cyber Requirements menu, we should see the import form, but this time the *remote_attestation* requirement will no longer be present because it was already added to the model. Select the *well_formed* requirement, enter a requirement ID (e.g., Req004_FlightPlanner), and click OK.

Similar to the *remote_attestation* requirement import, a *well_formed* Resolute prove statement is added to the Flight Planner, along with a new AGREE assume statement that evaluates FALSE (see Figure 27). The engineer will need to define what well-formed means in the context of incoming Flight Planner messages. In our case, well-formed means that the latitude, longitude and altitude values of the map waypoints are all within correct ranges, and the specified flight pattern is valid. We have added the formal definition of well-formed to the AGREE annex library at the bottom of the SW.aadl package.



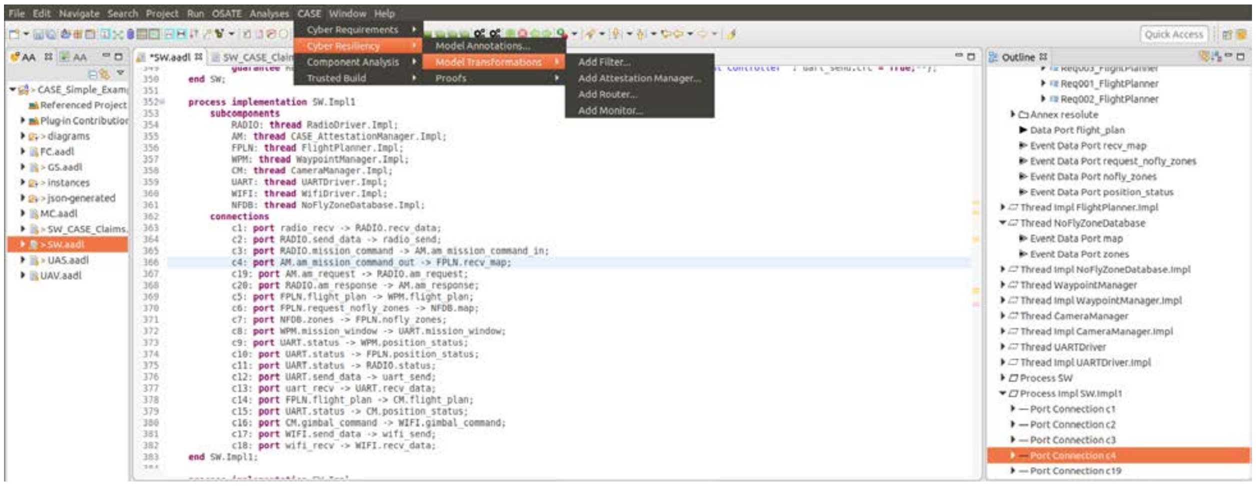
```
thread FlightPlanner
    -- The FlightPlanner is an abstraction for UxAS.
    -- It accepts a command message containing a map and flight pattern, and generates a mission.
    -- The FlightPlanner also has access to a No-Fly zone database, which it uses to generate the mission
    -- to avoid specified no-fly zones
    features
        flight_plan: out data port Mission.Impl;
        recv_map: in event data port Command.Impl;
        request_nofly_zones: out event data port Map.Impl;
        nofly_zones: in event data port MapArray.Impl;
        position_status: in event data port Coordinate.Impl;
    annex agree {**
        assume Req004_FlightPlanner "The FlightPlanner shall receive a well-formed command from the GroundStation" : FALSE;
        assume Req003_FlightPlanner "The FlightPlanner shall only accept messages from a trusted GroundStation" : TRUSTED_MESSAGE(recv_map);
        assume Req001_FlightPlanner "The Flight Planner shall receive a valid message from the Ground Station" : VALID_MESSAGE(recv_map);
        guarantee Req002_FlightPlanner "The Flight Planner shall generate a valid mission" : good_mission(flight_plan);**};
    annex resolute {**

        prove(well_formed(this, "Req004_FlightPlanner"))
        prove(remote_attestation(this, "Req003_FlightPlanner", RadioDriver, CASE_AttestationManager))
    **};
end FlightPlanner;
```

Figure 27. Flight Planner component after well_formed requirement import.

Replace the FALSE keyword with the WELL_FORMED_MESSAGE(recv_map) property, so the assume statement looks like the following:

```
assume Req004_FlightPlanner "The FlightPlanner shall receive a well-formed command from the GroundStation" : WELL_FORMED_MESSAGE(recv_map);
```

Running AGREE at this step will cause the analysis to fail because there isn't a component in the model that can guarantee the incoming command message is well-formed. To address this, we require another model transformation. This time we would like to insert a *filter* that will drop any malformed messages so they don't reach the Flight Planner. Whereas we selected a communication driver subcomponent in the process implementation for the Attestation Manager model transformation, for the filter we need to select the connection between two components where the filter should sit. In the SW.Impl1 implementation (make sure to work with the implementation that contains the Attestation Manager, and not the initial implementation), select connection c4, which connects the Attestation Manager to the Flight Planner. Next, select the CASE → Cyber Resiliency → Model Transformations → Add Filter… menu item, as shown in Figure 28.

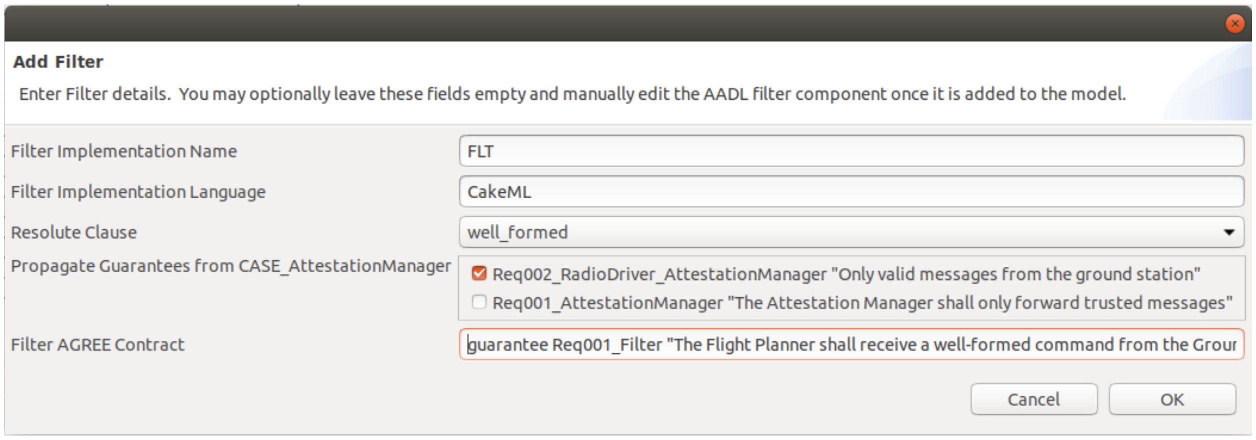


```
	end SW;

	process implementation SW.Impl1
		subcomponents
			RADIO: thread RadioDriver.Impl;
			AM: thread CASE_AttestationManager.Impl;
			FPLN: thread FlightPlanner.Impl;
			WPM: thread WaypointManager.Impl;
			CM: thread CameraManager.Impl;
			UART: thread UARTDriver.Impl;
			WIFI: thread WifiDriver.Impl;
			NFDB: thread NoFlyZoneDatabase.Impl;
		connections
			c1: port radio_recv -> RADIO.recv_data;
			c2: port RADIO.send_data -> radio_send;
			c3: port RADIO.mission_command -> AM.am_mission_command_in;
			c4: port AM.am_mission_command_out -> FPLN.recv_map;
			c19: port AM.am_request -> RADIO.am_request;
			c20: port RADIO.am_response -> AM.am_response;
			c5: port FPLN.flight_plan -> WPM.flight_plan;
			c6: port FPLN.request_nofly_zones -> NFDB.map;
			c7: port NFDB.zones -> FPLN.nofly_zones;
			c8: port WPM.mission_window -> UART.mission_window;
			c9: port UART.status -> WPM.position_status;
			c10: port UART.status -> FPLN.position_status;
			c11: port UART.status -> RADIO.status;
			c12: port UART.send_data -> uart_send;
			c13: port uart_recv -> UART.recv_data;
			c14: port FPLN.flight_plan -> CM.flight_plan;
			c15: port UART.status -> CM.position_status;
			c16: port CM.gimbal_command -> WIFI.gimbal_command;
			c17: port WIFI.send_data -> wifi_send;
			c18: port wifi_recv -> WIFI.recv_data;
	end SW.Impl1;
```

Figure 28. Adding a CASE Filter component to the model.

An *Add Filter* wizard will open. Enter the values as shown in Figure 29, and click OK. Be sure that the checkbox for propagating the Req001_AttestationManager guarantee is not checked. For the Filter AGREE contract, enter the following Guarantee statement (without line breaks):

guarantee Req001_Filter "The Flight Planner shall receive a well-formed command from the Ground Station" : WELL_FORMED_MESSAGE(filter_out) and (good_command(filter_out) => (filter_out = filter_in));


Add Filter

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

Filter Implementation Name	FLT
Filter Implementation Language	CakeML
Resolute Clause	well_formed
Propagate Guarantees from CASE_AttestationManager	[✓] Req002_RadioDriver_AttestationManager "Only valid messages from the ground station" [ ] Req001_AttestationManager "The Attestation Manager shall only forward trusted messages"
Filter AGREE Contract	guarantee Req001_Filter "The Flight Planner shall receive a well-formed command from the Grour

Figure 29. Add Filter wizard.

Save the model. The SW.aadl model will be modified to include a CASE_Filter component type and implementation. Figure 30 displays the CASE_Filter component type and implementation declarations. As with the previous model transformation, an entirely new SW implementation was generated, as shown in Figure 31. The new implementation (Figure 31b) contains a "FLT" subcomponent as well as the proper connection wiring. You may view the graphical model of this implementation by opening the SW_SW_Impl2.aadl_diagram file in the CASE_Simple_Example_V2/diagrams/ directory in the Navigation pane.

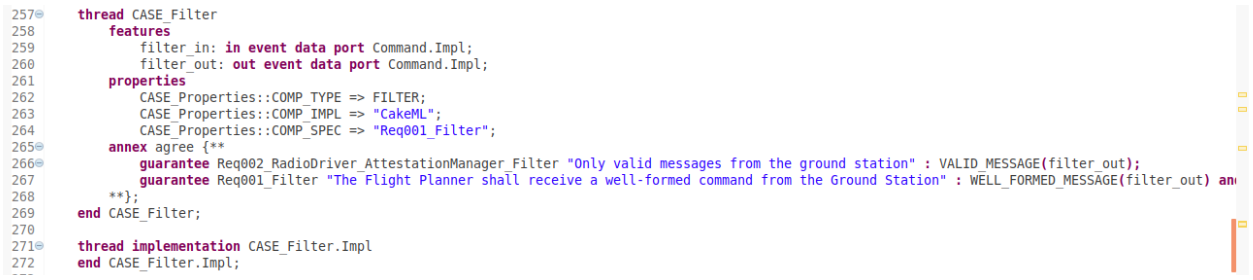
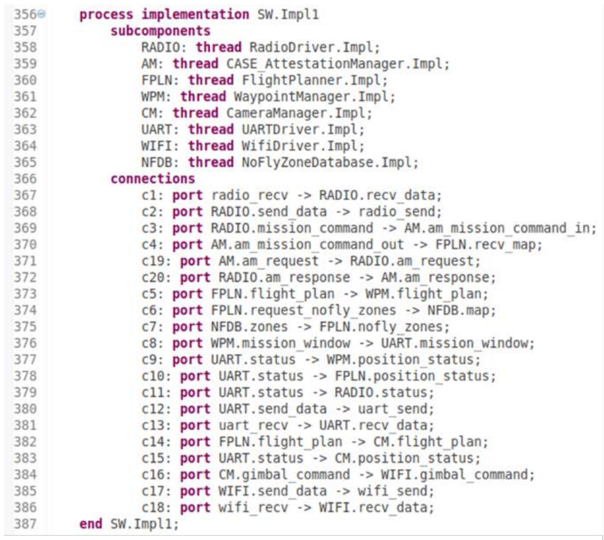
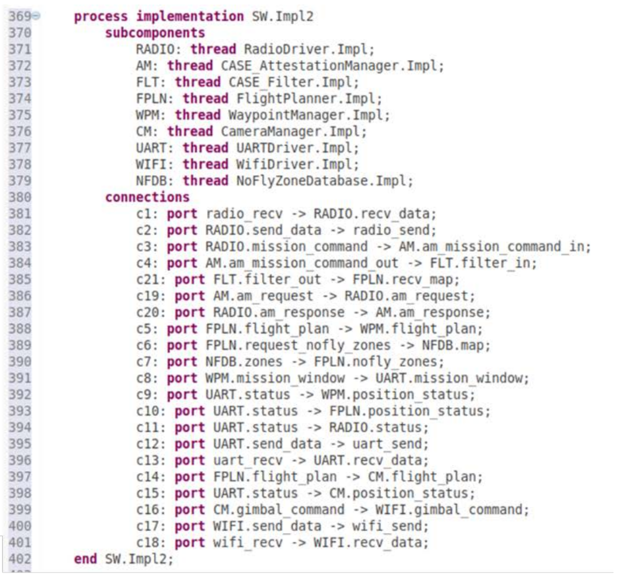

```
thread CASE_Filter
    features
        filter_in: in event data port Command.Impl;
        filter_out: out event data port Command.Impl;
    properties
        CASE_Properties::COMP_TYPE => FILTER;
        CASE_Properties::COMP_IMPL => "CakeML";
        CASE_Properties::COMP_SPEC => "Req001_Filter";
    annex agree {**
        guarantee Req002_RadioDriver_AttestationManager_Filter "Only valid messages from the ground station" : VALID_MESSAGE(filter_out);
        guarantee Req001_Filter "The Flight Planner shall receive a well-formed command from the Ground Station" : WELL_FORMED_MESSAGE(filter_out) an
    **};
end CASE_Filter;

thread implementation CASE_Filter.Impl
end CASE_Filter.Impl;
```

Figure 30. CASE_Filter component and implementation declaration.



```
process implementation SW.Impl1
    subcomponents
        RADIO: thread RadioDriver.Impl;
        AM: thread CASE_AttestationManager.Impl;
        FPLN: thread FlightPlanner.Impl;
        WPM: thread WaypointManager.Impl;
        CM: thread CameraManager.Impl;
        UART: thread UARTDriver.Impl;
        WIFI: thread WifiDriver.Impl;
        NFDB: thread NoFlyZoneDatabase.Impl;
    connections
        c1: port radio_recv -> RADIO.recv_data;
        c2: port RADIO.send_data -> radio_send;
        c3: port RADIO.mission_command -> AM.am_mission_command_in;
        c4: port AM.am_mission_command_out -> FPLN.recv_map;
        c19: port AM.am_request -> RADIO.am_request;
        c20: port RADIO.am_response -> AM.am_response;
        c5: port FPLN.flight_plan -> WPM.flight_plan;
        c6: port FPLN.request_nofly_zones -> NFDB.map;
        c7: port NFDB.zones -> FPLN.nofly_zones;
        c8: port WPM.mission_window -> UART.mission_window;
        c9: port UART.status -> WPM.position_status;
        c10: port UART.status -> FPLN.position_status;
        c11: port UART.status -> RADIO.status;
        c12: port UART.send_data -> uart_send;
        c13: port uart_recv -> UART.recv_data;
        c14: port FPLN.flight_plan -> CM.flight_plan;
        c15: port UART.status -> CM.position_status;
        c16: port CM.gimbal_command -> WIFI.gimbal_command;
        c17: port WIFI.send_data -> wifi_send;
        c18: port wifi_recv -> WIFI.recv_data;
end SW.Impl1;
```

(a)

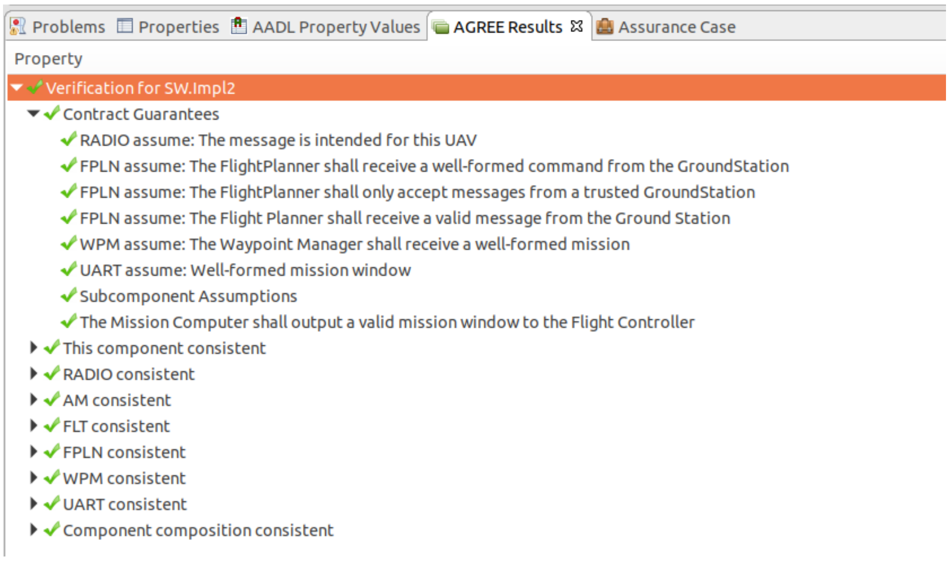


```
process implementation SW.Impl2
    subcomponents
        RADIO: thread RadioDriver.Impl;
        AM: thread CASE_AttestationManager.Impl;
        FLT: thread CASE_Filter.Impl;
        FPLN: thread FlightPlanner.Impl;
        WPM: thread WaypointManager.Impl;
        CM: thread CameraManager.Impl;
        UART: thread UARTDriver.Impl;
        WIFI: thread WifiDriver.Impl;
        NFDB: thread NoFlyZoneDatabase.Impl;
    connections
        c1: port radio_recv -> RADIO.recv_data;
        c2: port RADIO.send_data -> radio_send;
        c3: port RADIO.mission_command -> AM.am_mission_command_in;
        c4: port AM.am_mission_command_out -> FLT.filter_in;
        c21: port FLT.filter_out -> FPLN.recv_map;
        c19: port AM.am_request -> RADIO.am_request;
        c20: port RADIO.am_response -> AM.am_response;
        c5: port FPLN.flight_plan -> WPM.flight_plan;
        c6: port FPLN.request_nofly_zones -> NFDB.map;
        c7: port NFDB.zones -> FPLN.nofly_zones;
        c8: port WPM.mission_window -> UART.mission_window;
        c9: port UART.status -> WPM.position_status;
        c10: port UART.status -> FPLN.position_status;
        c11: port UART.status -> RADIO.status;
        c12: port UART.send_data -> uart_send;
        c13: port uart_recv -> UART.recv_data;
        c14: port FPLN.flight_plan -> CM.flight_plan;
        c15: port UART.status -> CM.position_status;
        c16: port CM.gimbal_command -> WIFI.gimbal_command;
        c17: port WIFI.send_data -> wifi_send;
        c18: port wifi_recv -> WIFI.recv_data;
end SW.Impl2;
```

(b)

Figure 31. Software implementations. (a) Software with Attestation Manager. (b) Software Attestation Manager and Filter.

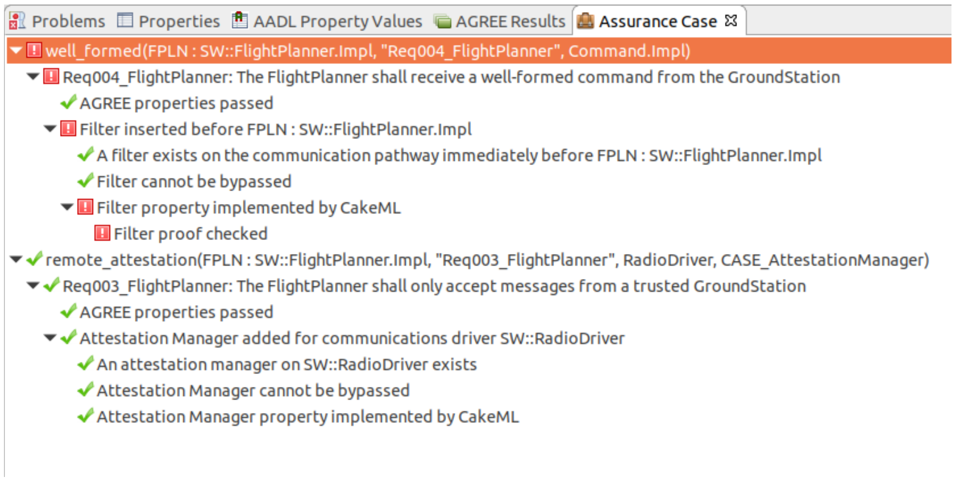
Make sure to save the model, then select the "Process Impl SW.Impl2" item in the Outline pane, and run AGREE. The properties should all pass, and the output should appear as in Figure 32.



```
✓ Verification for SW.Impl2
  ✓ Contract Guarantees
    ✓ RADIO assume: The message is intended for this UAV
    ✓ FPLN assume: The FlightPlanner shall receive a well-formed command from the GroundStation
    ✓ FPLN assume: The FlightPlanner shall only accept messages from a trusted GroundStation
    ✓ FPLN assume: The Flight Planner shall receive a valid message from the Ground Station
    ✓ WPM assume: The Waypoint Manager shall receive a well-formed mission
    ✓ UART assume: Well-formed mission window
    ✓ Subcomponent Assumptions
    ✓ The Mission Computer shall output a valid mission window to the Flight Controller
  ✓ This component consistent
  ✓ RADIO consistent
  ✓ AM consistent
  ✓ FLT consistent
  ✓ FPLN consistent
  ✓ WPM consistent
  ✓ UART consistent
  ✓ Component composition consistent
```

Figure 32. AGREE results with Attestation Manager and Filter component.

If we next run Resolute, we get output as shown in Figure 33. Notice that Resolute failed because it could not verify the filter proof.

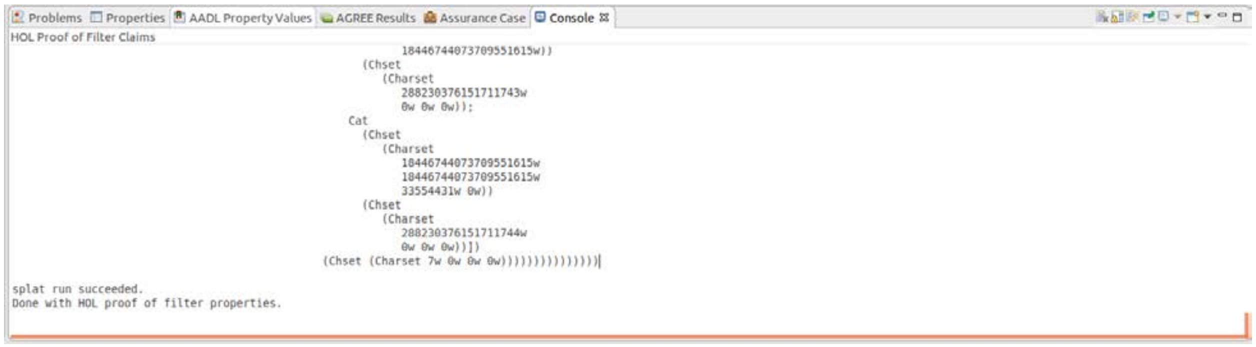


```
▼ ! well_formed(FPLN : SW::FlightPlanner.Impl, "Req004_FlightPlanner", Command.Impl)
  ▼ ! Req004_FlightPlanner: The FlightPlanner shall receive a well-formed command from the GroundStation
      ✓ AGREE properties passed
    ▼ ! Filter inserted before FPLN : SW::FlightPlanner.Impl
        ✓ A filter exists on the communication pathway immediately before FPLN : SW::FlightPlanner.Impl
        ✓ Filter cannot be bypassed
      ▼ ! Filter property implemented by CakeML
          ! Filter proof checked
▼ ✓ remote_attestation(FPLN : SW::FlightPlanner.Impl, "Req003_FlightPlanner", RadioDriver, CASE_AttestationManager)
  ▼ ✓ Req003_FlightPlanner: The FlightPlanner shall only accept messages from a trusted GroundStation
      ✓ AGREE properties passed
    ▼ ✓ Attestation Manager added for communications driver SW::RadioDriver
        ✓ An attestation manager on SW::RadioDriver exists
        ✓ Attestation Manager cannot be bypassed
        ✓ Attestation Manager property implemented by CakeML
```

Figure 33. Resolute could not verify the Filter proof.

In this demo, the CASE_Filter behavior is defined by an AGREE property, as indicated by the filter property COMP_SPEC (line 264 in Figure 30). The CakeML build tools for the CASE_Filter will guarantee that the regular expression implied by the AGREE property is correctly implemented. In addition, we also need to make sure this regular expression correctly implements the AGREE contract for the Filter. We have developed a tool called SPLAT (Semantic Properties for Language and Automata Theory) that supports the creation of, and correctness proofs for, filters specified by arithmetic properties extracted from AADL architectures. A record structure declaration with constraints on its fields specifies an encoding/decoding pair that maps a record of the given type into (and back out of) a sequence of bytes. A filter for such messages is also created, and embodied by a regular expression, which can be further compiled into a deterministic finite automata that checks that the sequence obeys the constraints. SPLAT extracts the filter properties from the architecture, creates encoders/decoders from the record declaration and accompanying constraints, creates a regular expression from the filter properties, and shows that the regular expression implements the filter properties. It produces a HOL theory capturing the formalization.

To perform this check, make sure the file SW.aadl is open in the OSATE editor and click the CASE → Cyber Resiliency → Proofs → SPLAT menu item. The result of the proof process is displayed in the OSATE console (see Figure 34).

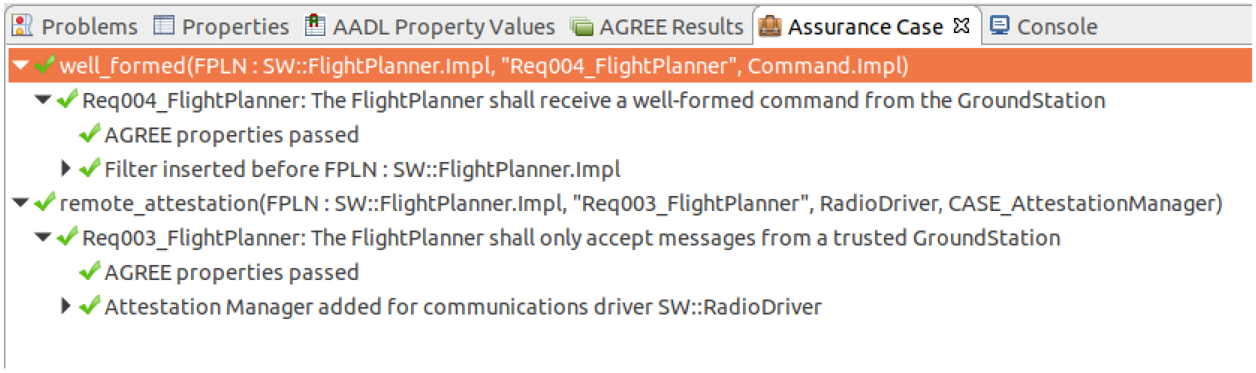


```
HOL Proof of Filter Claims
                                              18446744073709551615w))
                                         (Chset
                                            (Charset
                                              2882303761517117743w
                                              0w 0w 0w));
                                       Cat
                                         (Chset
                                            (Charset
                                              18446744073709551615w
                                              18446744073709551615w
                                              33554431w 0w))
                                         (Chset
                                            (Charset
                                              2882303761517117744w
                                              0w 0w 0w))])
                                    (Chset (Charset 7w 0w 0w 0w)))))))))))))))

splat run succeeded.
Done with HOL proof of filter properties.
```

Figure 34. HOL proof of filter claims

If the proof completes successfully, a certificate is generated, which will be associated with the CASE_Filter component in our SW model. This proof certificate is necessary evidence for the Resolute assurance case.

If the model has not been modified, re-running Resolute should produce the passing results shown in Figure 35. Otherwise, it may be necessary to re-run AGREE first.

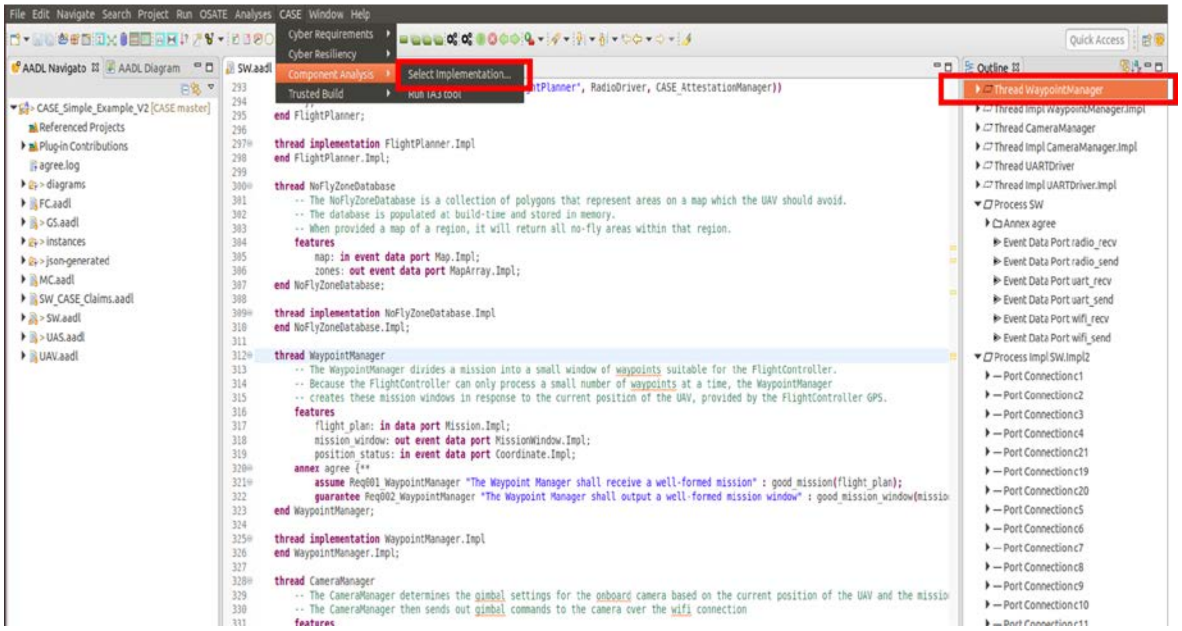


```
✓ well_formed(FPLN : SW::FlightPlanner.Impl, "Req004_FlightPlanner", Command.Impl)
    ✓ Req004_FlightPlanner: The FlightPlanner shall receive a well-formed command from the GroundStation
        ✓ AGREE properties passed
        ✓ Filter inserted before FPLN : SW::FlightPlanner.Impl
✓ remote_attestation(FPLN : SW::FlightPlanner.Impl, "Req003_FlightPlanner", RadioDriver, CASE_AttestationManager)
    ✓ Req003_FlightPlanner: The FlightPlanner shall only accept messages from a trusted GroundStation
        ✓ AGREE properties passed
        ✓ Attestation Manager added for communications driver SW::RadioDriver
```

Figure 35. Resolute results.

For components in our model that already have implementations, we would like to perform binary-level analysis using a TA3 tool. Such tools will require additional information about the component, such as the location of the binary on disk and perhaps the entry-function in code. Our tool provides an interface for entering this data into the model.

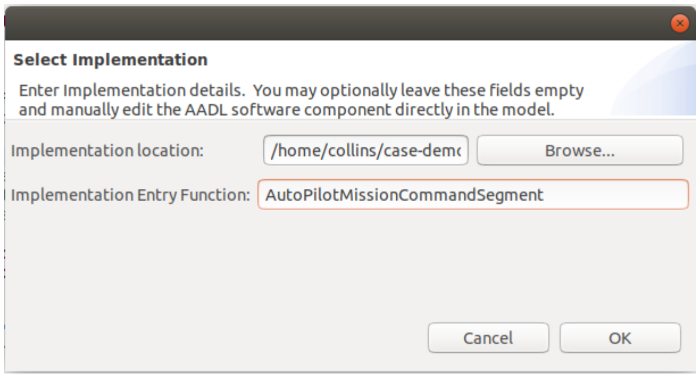
For example, suppose the Waypoint Manager is a legacy component for which an implementation exists. Select the Waypoint Manager component type declaration, as shown in Figure 36.



```
tPlanner", RadioDriver, CASE_AttestationManager))

end FlightPlanner;

thread implementation FlightPlanner.Impl
end FlightPlanner.Impl;

thread NoFlyZoneDatabase
    -- The NoFlyZoneDatabase is a collection of polygons that represent areas on a map which the UAV should avoid.
    -- The database is populated at build-time and stored in memory.
    -- When provided a map of a region, it will return all no-fly areas within that region.
    features
        map: in event data port Map.Impl;
        zones: out event data port MapArray.Impl;
end NoFlyZoneDatabase;

thread implementation NoFlyZoneDatabase.Impl
end NoFlyZoneDatabase.Impl;

thread WaypointManager
    -- The WaypointManager divides a mission into a small window of waypoints suitable for the FlightController.
    -- Because the FlightController can only process a small number of waypoints at a time, the WaypointManager
    -- creates these mission windows in response to the current position of the UAV, provided by the FlightController GPS.
    features
        flight_plan: in data port Mission.Impl;
        mission_window: out event data port MissionWindow.Impl;
        position_status: in event data port Coordinate.Impl;
    annex agree {**
        assume Req001_WaypointManager "The Waypoint Manager shall receive a well-formed mission" : good_mission(flight_plan);
        guarantee Req002_WaypointManager "The Waypoint Manager shall output a well-formed mission window" : good_mission_window(missio
end WaypointManager;

thread implementation WaypointManager.Impl
end WaypointManager.Impl;

thread CameraManager
    -- The CameraManager determines the gimbal settings for the onboard camera based on the current position of the UAV and the missio
    -- The CameraManager then sends out gimbal commands to the camera over the wifi connection
    features
```

Figure 36. Adding component implementation details for TA3 analysis.

The wizard as pictured in Figure 37 will open.



Select Implementation

Enter Implementation details. You may optionally leave these fields empty and manually edit the AADL software component directly in the model.

Implementation location: /home/collins/case-demo

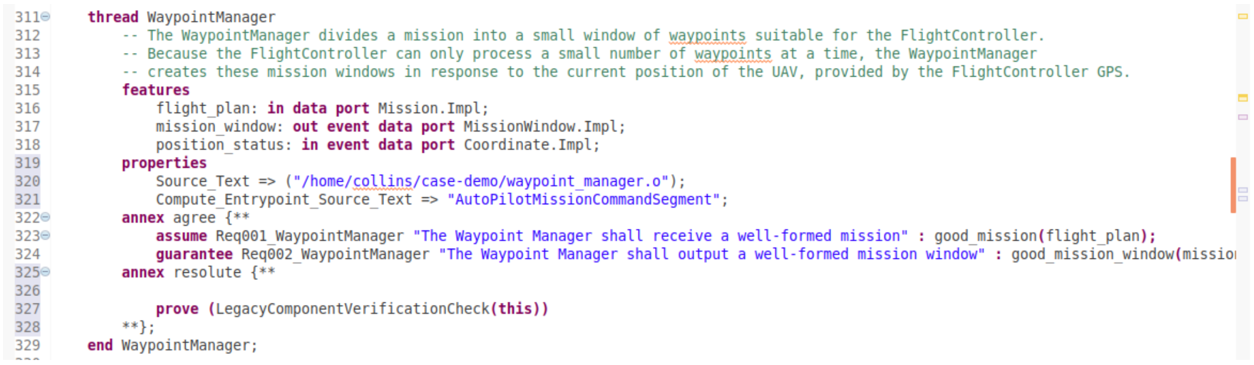
Implementation Entry Function: AutoPilotMissionCommandSegment

Figure 37. Select Implementation input wizard.

For this demo, to enter the location of the binary, we can click Browse and navigate to

/home/collins/case-demo

and select the **waypoint_manager.o** file. For the Implementation Entry Function, enter **AutoPilotMissionCommandSegment**. Click OK and the Waypoint Manager component will be annotated with the implementation information, as well as a Resolute clause for assuring that the TA3 check has been performed (see Figure 38). Our TA2 tool is not yet integrated with any TA3 tools, so there is currently no way to provide evidence for this claim.



```
thread WaypointManager
    -- The WaypointManager divides a mission into a small window of waypoints suitable for the FlightController.
    -- Because the FlightController can only process a small number of waypoints at a time, the WaypointManager
    -- creates these mission windows in response to the current position of the UAV, provided by the FlightController GPS.
    features
        flight_plan: in data port Mission.Impl;
        mission_window: out event data port MissionWindow.Impl;
        position_status: in event data port Coordinate.Impl;
    properties
        Source_Text => ("/home/collins/case-demo/waypoint_manager.o");
        Compute_Entrypoint_Source_Text => "AutoPilotMissionCommandSegment";
    annex agree {**
        assume Req001_WaypointManager "The Waypoint Manager shall receive a well-formed mission" : good_mission(flight_plan);
        guarantee Req002_WaypointManager "The Waypoint Manager shall output a well-formed mission window" : good_mission_window(missio
    annex resolute {**

        prove (LegacyComponentVerificationCheck(this))
    **};
end WaypointManager;
```

Figure 38. Waypoint Manager component annotated with implementation information.

Demonstration of the High-Assurance Filter Component

To demonstrate the results of the build process (still under development), 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 39). The producer emits strings to the CakeML Filter, which matches them against a regular expression and either forwards them 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.

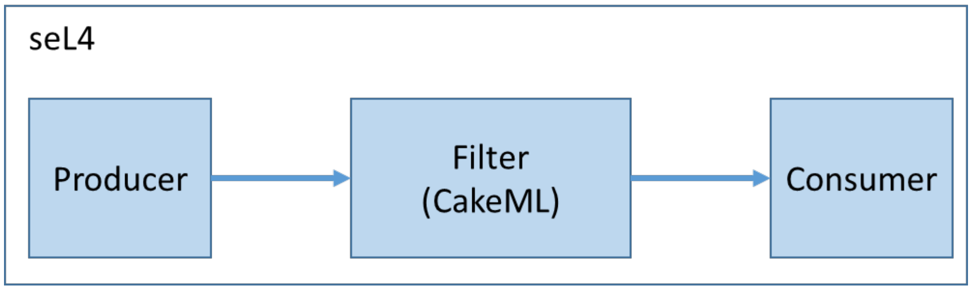


Figure 39. Filter test setup.

A CAmkES system consists of several *components*, communicating via *connections*. In our example system, our three components communicate via two connections, configured in a CAmkES file. The 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

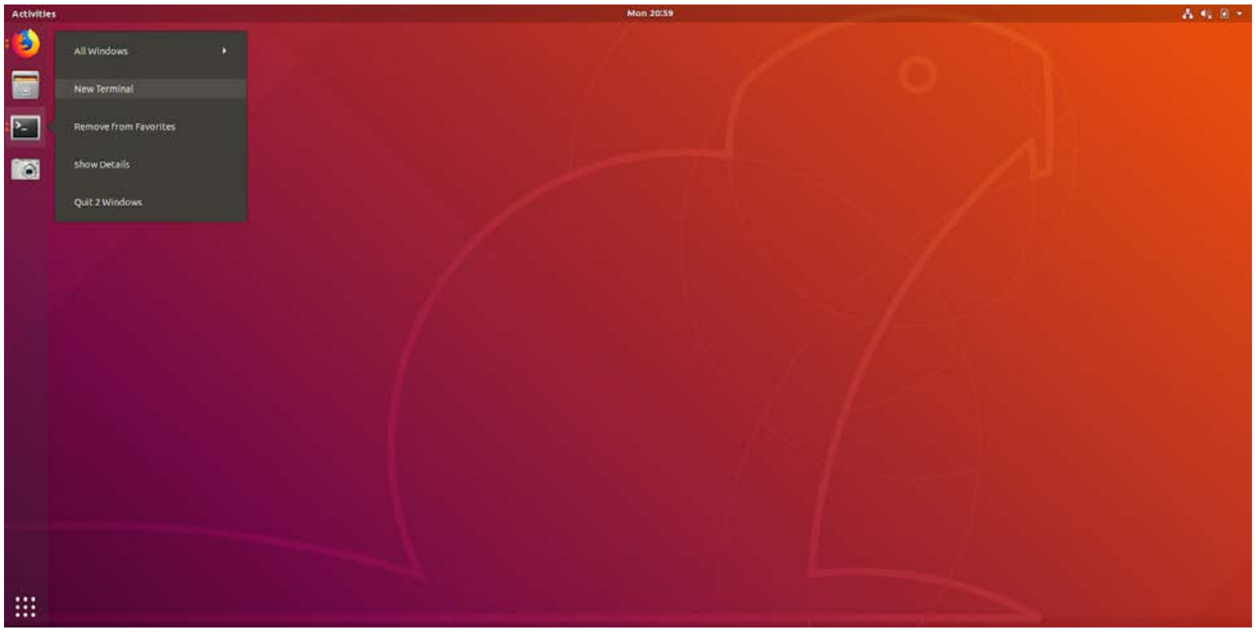
https://github.com/seL4/camkes-tool/blob/master/docs/index.md

Since the first TA7 tool assessment, several improvements have been made to the CakeML filter application. First, the CAmkES tool has been extended with native support for components written in CakeML. This allowed us to remove the complicated C code which was responsible for interfacing the application logic with the C code that CAmkES previously generated for communication between components. In the new version of the application, CAmkES generates a CakeML event loop which calls seL4 system calls directly, and dispatches remote procedure calls to the correct CakeML functions.

Furthermore, the handling of strings has been improved by our transition to the new version of CAmkES. Previously we used a fixed-size buffer to hold the input to the filter, which meant that the filter regular expression had to account for the trailing sequence of null bytes. In our improved version of the application, this is no longer required, as input strings are correctly truncated so that they contain no extraneous characters.

To demo the high-assurance filter component, perform the following steps:

1. Open a new terminal by right clicking on the terminal on the favorites bar and selecting "New Terminal"



2. Next, type
   cd ~/case-demo/cakeml-regex-filter/
   then hit enter.
3. Next, type
   (rm -rf build && mkdir build && cd build && ../init-build.sh -DCAKEMLDIR=/home/collins/case-demo/cakeml -DCAMKES_APP=cakeml_regex -DFilterRegex="This.*" && ninja)
   (without line breaks) then hit enter. You will see a lot of scrolling text; these are compilation messages and can be safely ignored. Note that the compilation may take several minutes to complete.
4. Next, type
   (cd build && ./simulate)
   then hit enter. This runs the application compiled by CAmkES. The output displays messages which pass the filter and should look something like this:

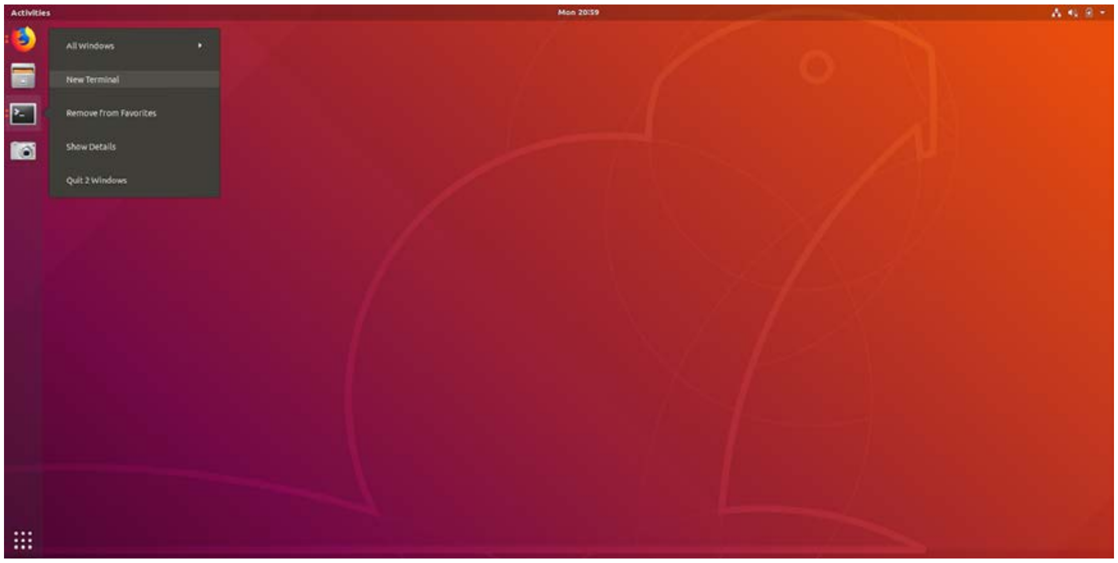
```
This will get through 1
This will get through 2
This will get through 3
This will get through 1
This will get through 2
This will get through 3
This will get through 1
This will get through 2
This will get through 3
This will get through 1
This will get through 2
This will get through 3
```

5. To filter different messages, the regular expression in Step 3 can be changed.
6. Close the terminal to kill the process.

Demonstration of the Remote Attestation Component

A remote attestation implementation for the scenario identified in our model has been developed. A demo of the functionality can be run by performing the following steps. Note that running 'make clean' at any time should restore the demo files to a good, predictable initial state (while avoiding a complete re-build).

1. Open a new terminal by right clicking on the terminal icon in the favorites bar and selecting "New Terminal.



2. Type
   cd ~/case-demo/caseAttestationDemo/
   then hit enter.
3. Type
   make run
   then hit enter. This command invokes the attestation protocol interpreter, providing as input a protocol term and initial evidence. The interpreter then outputs the result (evidence + appraisal decision) of protocol execution. Most of the input/output is printed to the terminal (except for the JSON data exchange objects, which the tool outputs to a file). The protocol and evidence terms are written in an evolving attestation protocol specification language called Copland. See the "Output Files" section below for more detailed descriptions of these Copland terms.
4. Type
   make attack
   then hit enter. This command invokes a script that modifies the "file of interest" in this attestation protocol, a file named "target.txt" in the current directory.
5. Type
   make run
   then hit enter. This will again invoke the interpreter, asking it to re-run the protocol. This time, however, it detects the attack on "target.txt". You'll notice that the Appraisal Result section of

the terminal output reports that the hash of "target.txt" does not match the golden (expected) value.
Type
make repair
then hit enter. This command will restore "target.txt" to its original (good) contents.

6. Type
   make run
   then hit enter. Confirm that the protocol successfully appraises the patched "target.txt".

The tool output is contained in the following files:

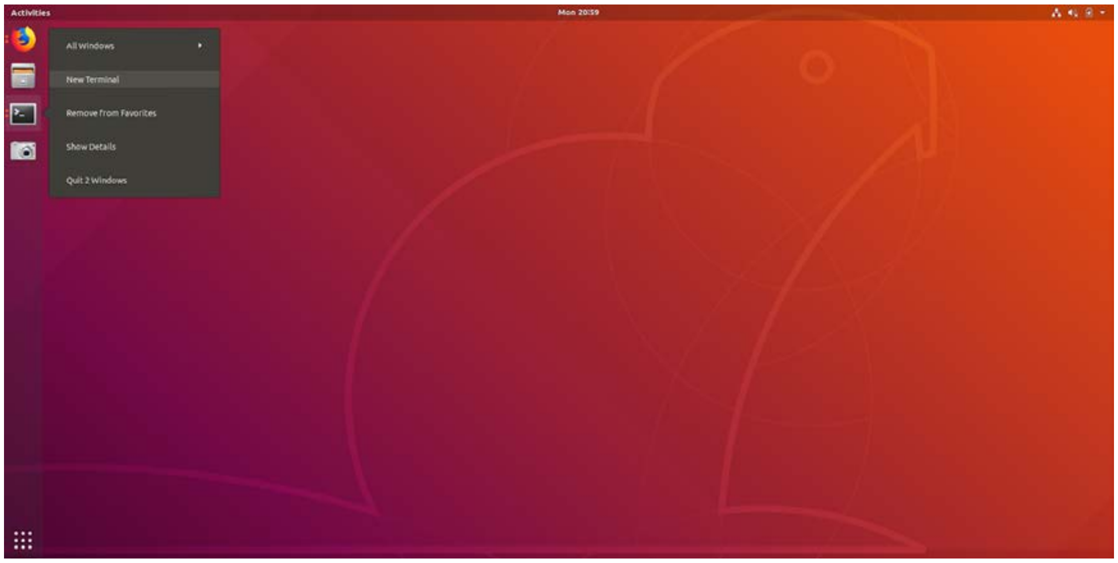
1. **demoOutput/protoIn.hs**:
   This file contains abstract syntax representations of the input protocol term and initial evidence. LN takes two protocol terms and performs them in sequence, NONCE generates a nonce (random number), SIG performs a digital signature over the accumulated evidence bundle, and (USM 1) is a user-space measurement that performs a SHA-256 hash of its filename argument. The protocol term has the form:
   - Perform in sequence:
     - Generate a nonce
     - Send a remote attestation request to place 1 (passing the nonce as initial evidence), asking it to perform the following protocol:
       - Perform these actions in sequence:
         - Take a SHA-256 hash of the file "target.txt" and append the result to the nonce
         - Sign the resulting bundle with your private key.

2. **demoOutput/protoOut.hs:**
   This file contains an evidence term (result of protocol execution). *G* represents a "signed bundle", holding the payload and the signature. The payload in this case is a *U* (result of the file hash USM), with a nested *N* (the nonce value). Hash values, nonce values, and the signature bits are shortened with "..." in the output for readability purposes.
3. **demoOutput/jsonIn.hs:**
   This file contains JSON representations of the terms corresponding to the input protocol and initial evidence from demoOutput/protoIn.hs. A canonical JSON data exchange format for protocol terms and evidence is crucial for coordinating attestations amongst diverse platforms.
4. **demoOutput/jsonOut.hs:**
   This file contains the JSON representation of the evidence term resulting from protocol execution.

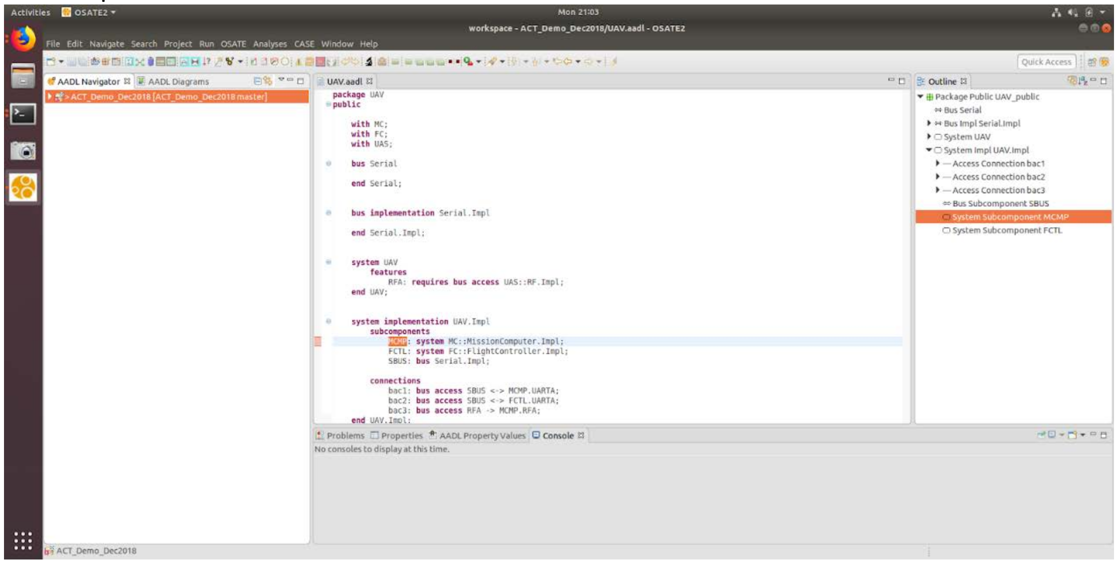
Demonstration of the Initial System Build Tool

Once we have implementations for the components in our system, we can build using the ACT tool. Although the ACT tool has been packaged as an OSATE plugin, this demonstration uses a different OSATE build as well as a slightly different model than we used to demonstrate the Model Transformation functionality. We will push for convergence of both in the near future. For now the demo can be run by performing the following steps.

3. Open a new terminal by right clicking on the terminal on the favorites bar and selecting “New Terminal”.



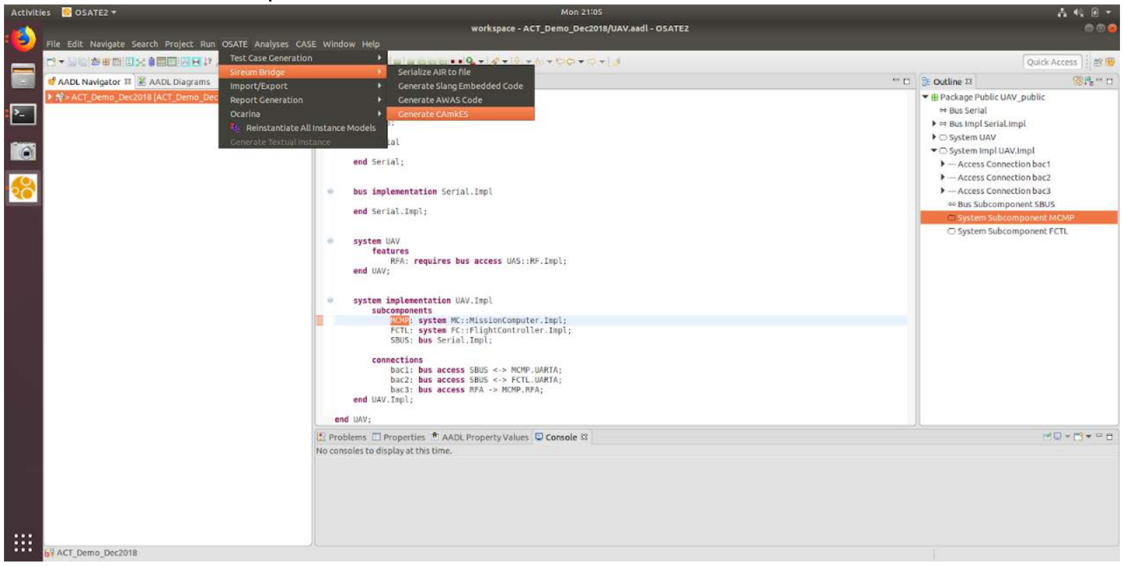
4. Type
   cd ~/case-demo/osate2-2.4.0-v20181210-2201-linux.gtk.x86_64/
   then hit enter.
5. Type
   ./osate
   then hit enter.
6. Expand the “System Impl UAV.Impl” in the “Outline” frame on the right and highlight “System Subcomponent MCMP”.



```
package UAV
public

    with MC;
    with FC;
    with UAS;

    bus Serial

    end Serial;


    bus implementation Serial.Impl

    end Serial.Impl;


    system UAV
        features
            RFA: requires bus access UAS::RF.Impl;
    end UAV;


    system implementation UAV.Impl
        subcomponents
            MCMP: system MC::MissionComputer.Impl;
            FCTL: system FC::FlightController.Impl;
            SBUS: bus Serial.Impl;

        connections
            bac1: bus access SBUS <-> MCMP.UARTA;
            bac2: bus access SBUS <-> FCTL.UARTA;
            bac3: bus access RFA -> MCMP.RFA;
    end UAV.Impl;
```

```
No consoles to display at this time.
```

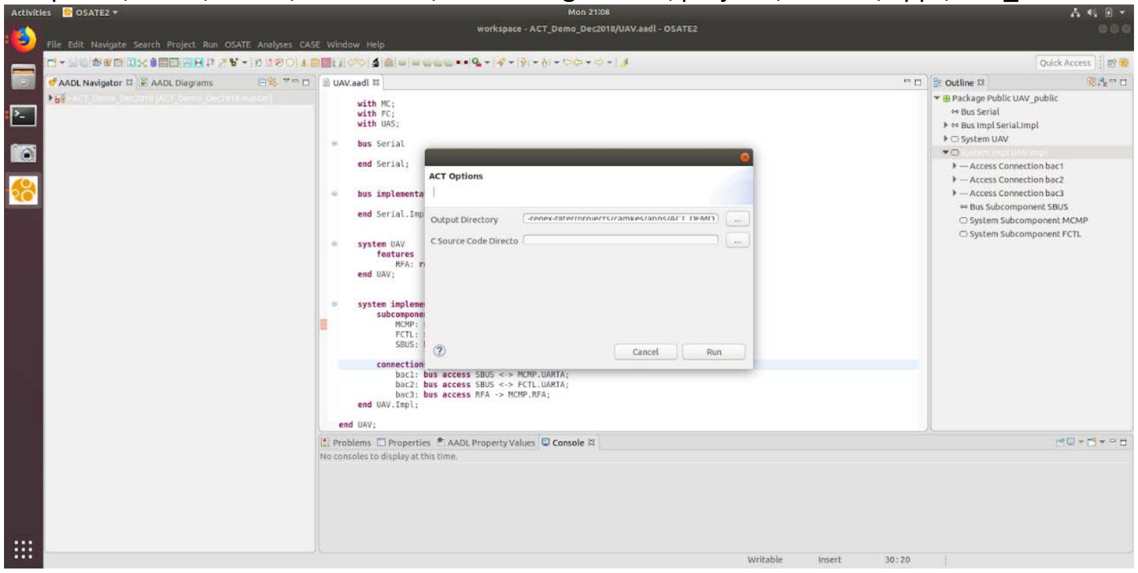
7. Open the “OSATE” dialog menu, navigate to the “Sireum Bridge” dialog menu and click on the “Generate CAmkES” option.



```
    end Serial;

    bus implementation Serial.Impl

    end Serial.Impl;

    system UAV
        features
            RFA: requires bus access UAS::RF.Impl;
    end UAV;

    system implementation UAV.Impl
        subcomponents
            MCMP: system MC::MissionComputer.Impl;
            FCTL: system FC::FlightController.Impl;
            SBUS: bus Serial.Impl;

        connections
            bac1: bus access SBUS <-> MCMP.UARTA;
            bac2: bus access SBUS <-> FCTL.UARTA;
            bac3: bus access RFA -> MCMP.RFA;
    end UAV.Impl;

end UAV;
```

No consoles to display at this time.

8. This will bring up a menu that will indicate output from the generation process will be placed at the path “/home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO”.



```
    with MC;
    with FC;
    with UAS;

    bus Serial

    end Serial;

    bus implementa

    end Serial.Imp

    system UAV
        features
            RFA: r
    end UAV;

    system impleme
        subcompone
            MCMP:
            FCTL:
            SBUS:

        connection
            bac1: bus access SBUS <-> MCMP.UARTA;
            bac2: bus access SBUS <-> FCTL.UARTA;
            bac3: bus access RFA -> MCMP.RFA;
    end UAV.Impl;

end UAV;
```

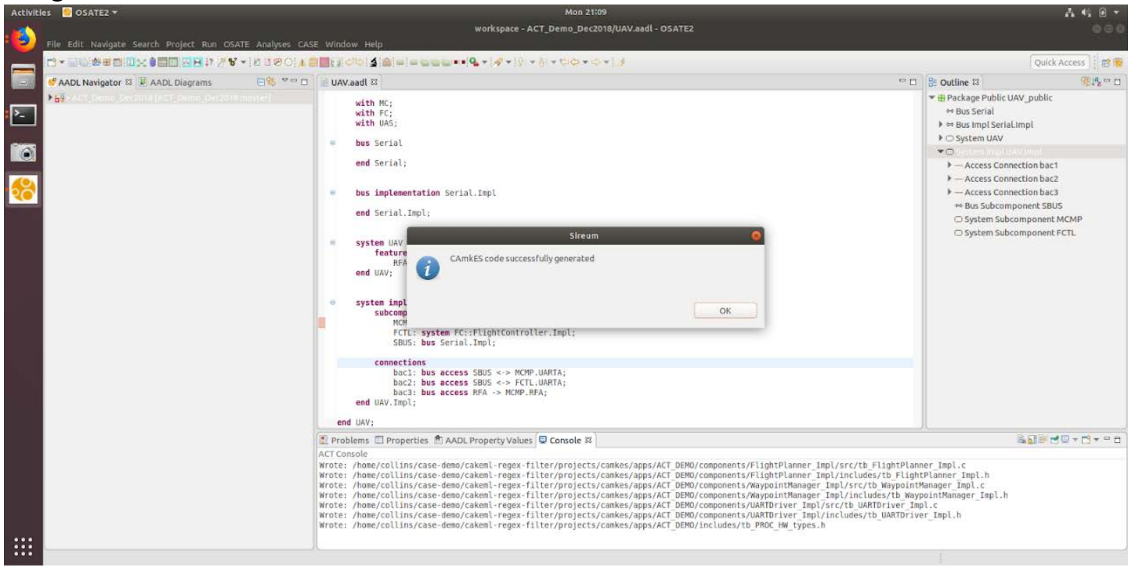
ACT Options

Output Directory: -regex-filter/projects/camkes/apps/ACT_DEMO

C Source Code Directo:

No consoles to display at this time.

9. Click on the "Run" button. This should produce some output and a pop-up window indicating the generation was successful.



```
with MC;
with FC;
with UAS;

bus Serial

end Serial;


bus implementation Serial.Impl

end Serial.Impl;


system UAV
    feature
        RFA
end UAV;


system impl
    subcomp
        MCM
        FCTL: system FC::FlightController.Impl;
        SBUS: bus Serial.Impl;

    connections
        bac1: bus access SBUS <-> MCMP.UARTA;
        bac2: bus access SBUS <-> FCTL.UARTA;
        bac3: bus access RFA -> MCMP.RFA;
end UAV.Impl;

end UAV;
```

CAmkES code successfully generated

```
ACT Console
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/FlightPlanner_Impl/src/tb_FlightPlanner_Impl.c
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/FlightPlanner_Impl/includes/tb_FlightPlanner_Impl.h
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/WaypointManager_Impl/src/tb_WaypointManager_Impl.c
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/WaypointManager_Impl/includes/tb_WaypointManager_Impl.h
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/UARTDriver_Impl/src/tb_UARTDriver_Impl.c
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/components/UARTDriver_Impl/includes/tb_UARTDriver_Impl.h
Wrote: /home/collins/case-demo/cakeml-regex-filter/projects/camkes/apps/ACT_DEMO/includes/tb_PROC_HW_types.h
```

10. Click "OK" and exit from OSATE via the "FILE" menu.
11. Return to the terminal you opened earlier and type
    cd ~/case-demo/cakeml-regex-filter/
    then hit enter.
12. Type:
    (rm -rf build_act_demo && mkdir build_act_demo && cd build_act_demo && ../init-build.sh -DCAMKES_APP=ACT_DEMO && ninja)
    (without line breaks) then hit enter. You will see a lot of scrolling text; these are compilation messages and can be safely ignored.
13. Enter
    (cd build_act_demo && ./simulate)
    then hit enter. This runs the application compiled by CAmkES and generated by the ACT tool. The messages describe the interactions between the components in the application and should look something like the following:

```
RDIO:> Sending command.
FPLN:< Command.
FPLN:> New mission notification.
WM:< New mission notification.
WM:> Received mission notification.
UART:< Packet.
UART:< Packet.
UART:< Packet.
UART:< Packet.
FPLN:< Received mission notification.
UART:> Sending 2 as the next id.
WM:< Received 2 as the next id.
UART:< Packet.
UART:< Packet.
UART:< Packet.
UART:< Packet.
UART:> Sending 3 as the next id.
```

```
WM:< Received 3 as the next id.
UART:< Packet.
UART:< Packet.
UART:< Packet.
UART:< Packet.
UART:> Sending 4 as the next id.
WM:< Received 4 as the next id.
```