Collins Aerospace

Simple UAV Example

BriefCASE Workflow Illustration

June 18, 2021

1 Overview

The Simple UAV Example is intended to illustrate an end-to-end application of the BriefCASE cyber-resiliency tools and workflow. It includes basic implementation code for the non-hardened components, and external inputs and outputs have been eliminated so that the hardened system can be generated and executed on QEMU. The goal is to show how the tools are used and provide a starting point for experimentation and evaluation of the tool capabilities. A more detailed description of the BriefCASE tool capabilities is found in the User Guide.

This document will step through the BriefCASE workflow, starting with an initial AADL model and building the initial version of the system from the model. Next, we will import cyber requirements to be addressed and transform the model to mitigate the cyber vulnerabilities corresponding to those requirements. If desired, the AGREE and Resolute tools can be run on the transformed system to demonstrate that the requirements have been satisfied in the model. Finally, we will generate code from the hardened model, and run the final system on QEMU.

The model and code can be found in the Loonwerks github repository:

https://github.com/loonwerks/CASE/tree/master/CASE_Simple_Example_V4

They are also provided with the CASE virtual machine tool environment (configured using the Vagrant script), located in the ~/CASE/uav-example/ directory.

2 Initial Model

We start with a basic UAV application consisting of three components:

- Radio periodically produces MissionCommand messages consisting of a node ID and two waypoints (initial and final points) describing a desired flight plan. Some messages will contain a bad ID (to test attestation) and some messages will contain a malformed waypoint (to test filtering).
- FlightPlanner receives MissionCommand messages from the Radio, inserts a new waypoint midway between the initial and final points, and produces the resulting FlightPlan message. Some messages will contain a bad middle waypoint to test monitoring.
- FlightController prints the received FlightPlan messages to the console and is also able to display alerts generated by a monitor.

The initial model is shown in Figure 1.

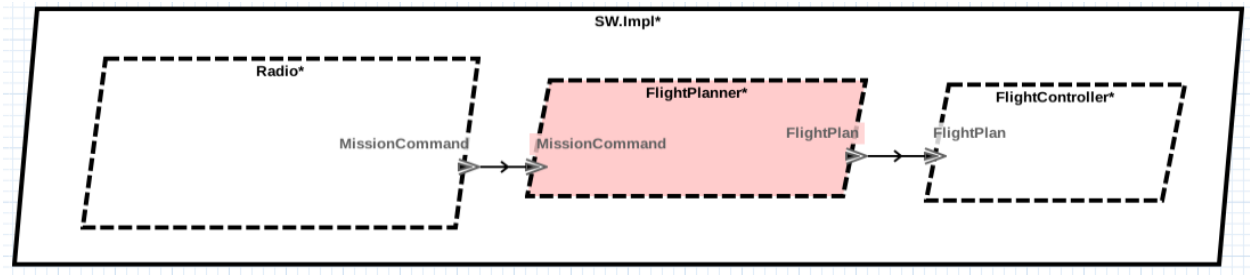


Figure 1. Initial AADL model.

3 Building and Running the Initial Model

The initial model can be built and run for either Linux or seL4. Since Linux does not have a real-time scheduler, the periodic Radio component will only run once and output a single MissionCommand message. If you build for seL4, the Radio will run periodically and output a new MissionCommand once per second. We will describe how to do that later in this document.

The AADL project is laid out as shown in Figure 2.

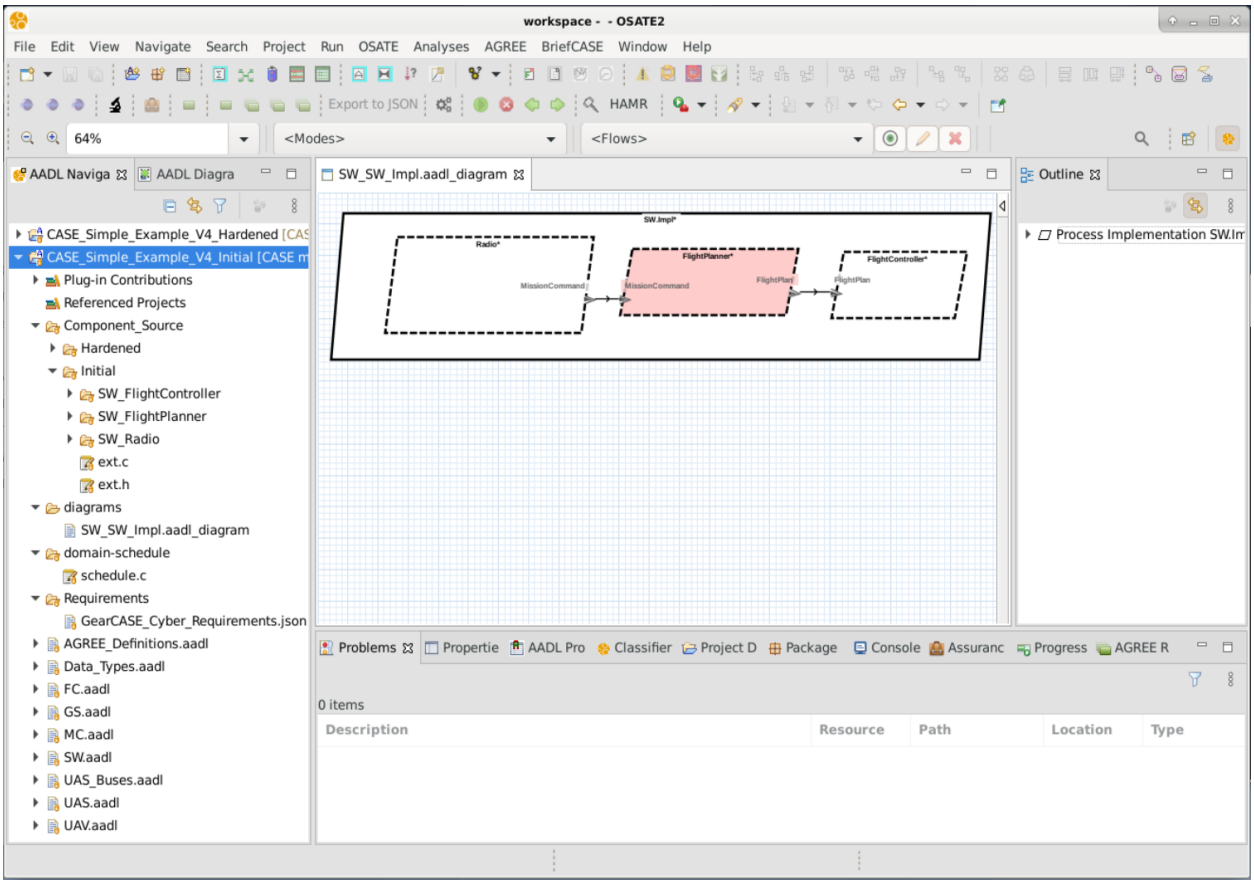


Figure 2. Simple UAV Example AADL project.

The SW.aadl file contains the software components, and the MC.aadl file defines the Mission Computer implementation containing the software components listed above, as well as the processor they will run on. For this example, we will build the Initial model to run on Linux.

We run HAMR on the Mission Computer by selecting MissionComputer.Impl in the Outline pane (see Figure 3) and clicking the HAMR button in the OSATE toolbar.

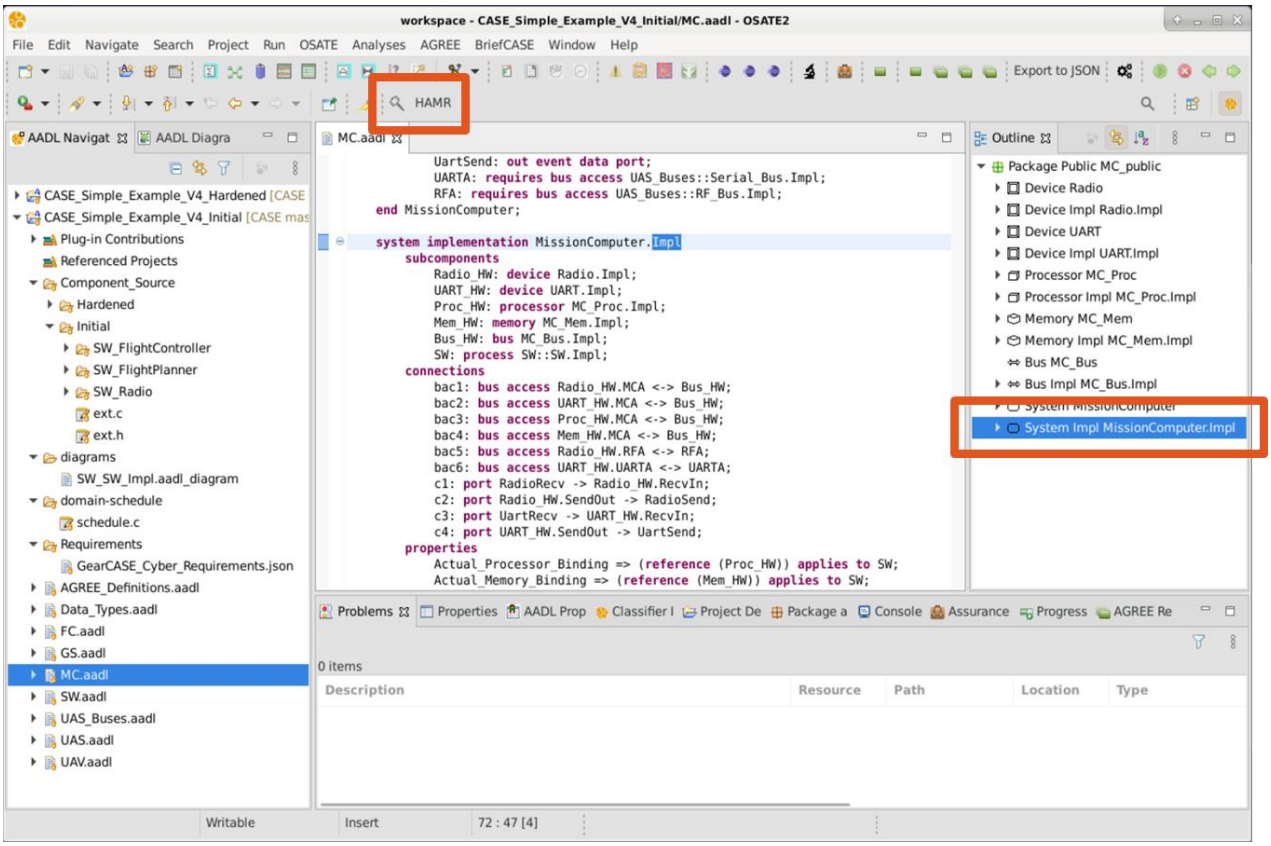


Figure 3. Running HAMR on the Mission Computer implementation.

In the HAMR dialog box that appears, specify the output directory, which we refer to as *HAMR_ROOT* from here on. For this example, the output directory must be named *HAMR_Simple_V4* to avoid linking errors with the included component implementations. Exclude Slang components in the dialog box and select 64-bit width, as shown in Figure 4, before clicking OK.

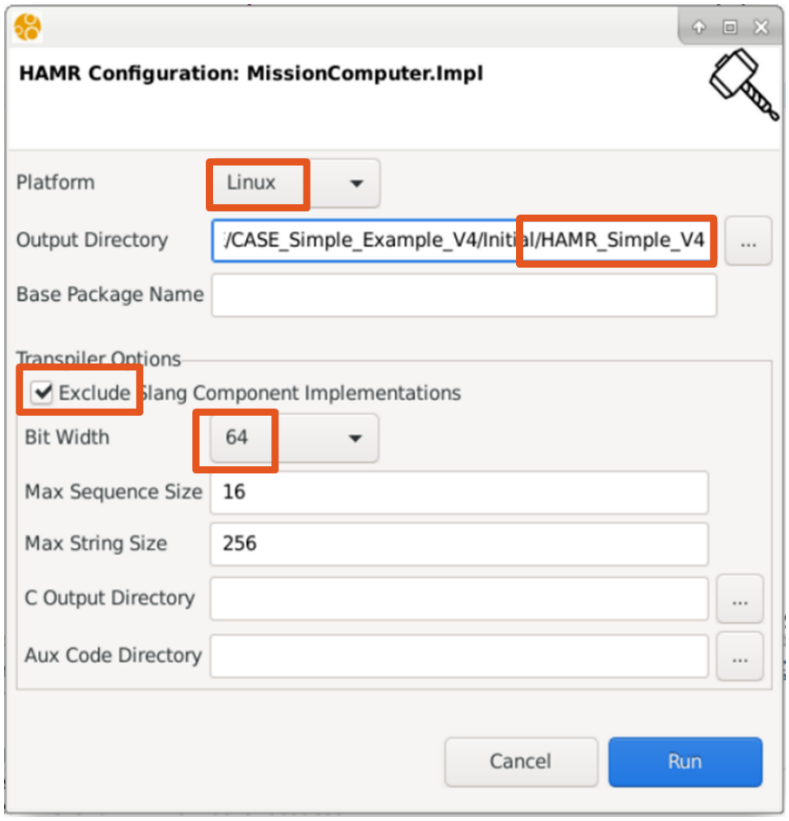


Figure 4. HAMR configuration dialog.

HAMR should then produce output as shown in Figure 5.

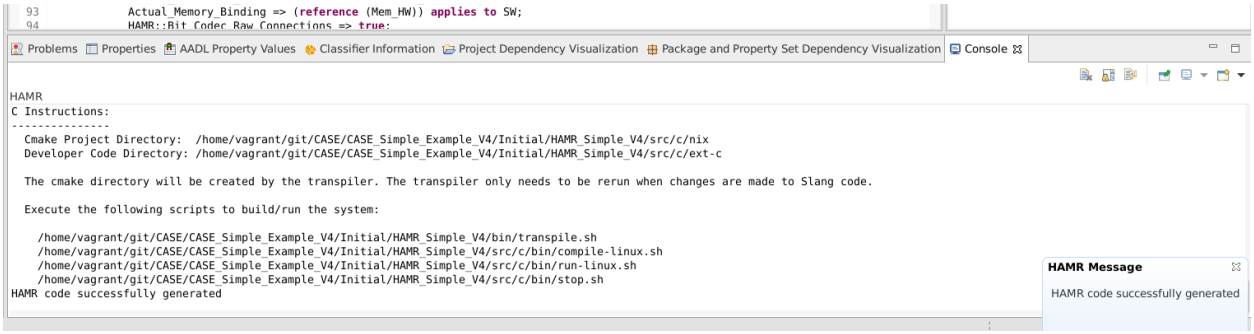


Figure 5. HAMR code generation status.

Once HAMR has been run successfully, the following directory structure is generated within HAMR_ROOT:

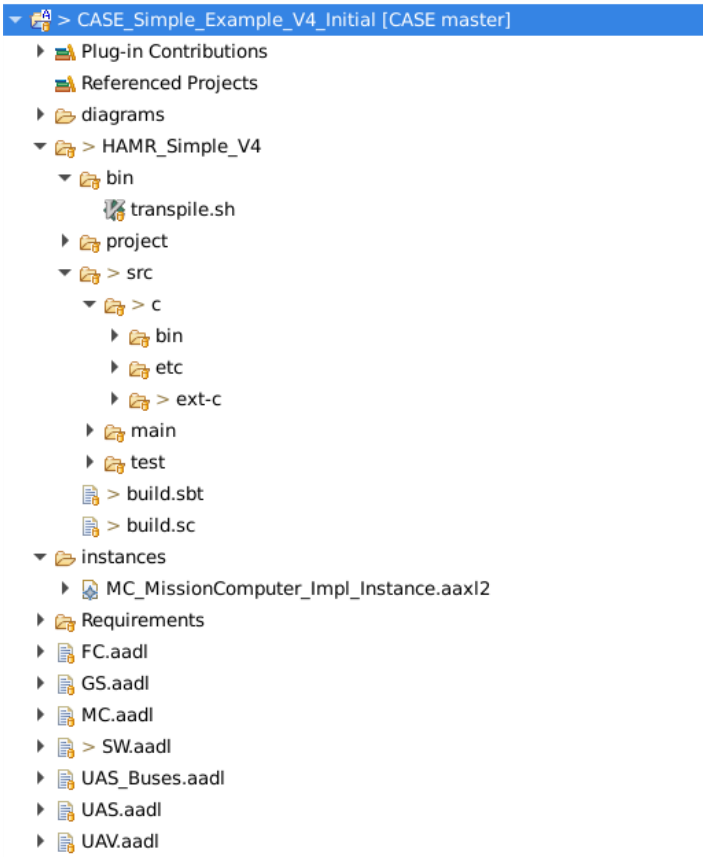


Figure 6. HAMR directory structure.

Next, run the HAMR transpiler from the command prompt as follows:

```
./HAMR_Simple_V4/bin/transpile.sh
```

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Initial$ ls
diagrams  FC.aadl  GS.aadl  HAMR_Simple_V4  instances  MC.aadl  Requirements  SW.aadl  UAS.aadl  UAS_Buses.aadl  UAV.aadl
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Initial$ ./HAMR_Simple_V4/bin/transpile.sh
Reading extension files ...
Time: 500 ms, Memory: 14.22 MB

Reading cmake include files ...
Reading sourcepath files ...
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/DataContent.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/ArtDebug.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/ArtTimer.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/ArtNative.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/Art.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/art/ArchitectureDescription.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/data/HAMR_Simple_V4/Base_Types.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/architecture/HAMR_Simple_V4/Arch.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/FlightPlanner_Impl_Api.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/AttestationTester_Impl_Api.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/FlightPlanner_Impl_SW_FlightPlanner_Bridge.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/FlightController_Impl_Api.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/AttestationTester_Impl_SW_AttestationTester_Bridge.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/RadioDriver_Impl_SW_Radio_Bridge.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/RadioDriver_Impl_Api.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/bridge/HAMR_Simple_V4/SW/FlightController_Impl_SW_FlightController_Bridge.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/Process.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/RadioDriver_Impl_SW_Radio_App.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/Platform.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/ArtNix.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/SharedMemory.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/AttestationTester_Impl_SW_AttestationTester_App.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/Main.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/PlatformNix.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/FlightController_Impl_SW_FlightController_App.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/IPC.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/nix/HAMR_Simple_V4/FlightPlanner_Impl_SW_FlightPlanner_App.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/component/HAMR_Simple_V4/SW/FlightPlanner_Impl_SW_FlightPlanner.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/component/HAMR_Simple_V4/SW/FlightController_Impl_SW_FlightController.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/component/HAMR_Simple_V4/SW/RadioDriver_Impl_SW_Radio.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/component/HAMR_Simple_V4/SW/AttestationTester_Impl_SW_AttestationTester.scala
Read /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/main/component/HAMR_Simple_V4/TranspilerToucher.scala
Time: 118 ms, Memory: 16.72 MB

Parsing, resolving, type outlining, and type checking Slang library files ...
```

```
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/FlightContro
ler.h to /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/FlightController_Impl_SW_FlightC
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/FlightContro
ler_api.h to /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/FlightController_Impl_SW_Fli
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/FlightContro
ler.c to /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/FlightController_Impl_SW_FlightC
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/ext.h to /ho
c/c/ext-c/ext.h
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/RadioDriver_
ASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/RadioDriver_Impl_SW_Radio/RadioDriver_Impl_SW_Radio_api.c
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/RadioDriver_
CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/RadioDriver_Impl_SW_Radio/RadioDriver_Impl_SW_Radio.c
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/RadioDriver_
CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/RadioDriver_Impl_SW_Radio/RadioDriver_Impl_SW_Radio.h
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/ext-c/RadioDriver_
ASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/ext-c/RadioDriver_Impl_SW_Radio/RadioDriver_Impl_SW_Radio_api.h
Created symlink from /home/vagrant/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c/nix/ext/etc/ipc.c to /home
c/etc/ipc.c
Time: 1801 ms, Memory: 190.30 MB

Ok! Total time: 27.97 s, Max memory: 311.28 MB
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Initial$
```

After transpiling, the behavior code for the Radio, FlightPlanner and FlightController can be inserted. The directory structure is shown in Figure 7 and the source files to be modified are highlighted.

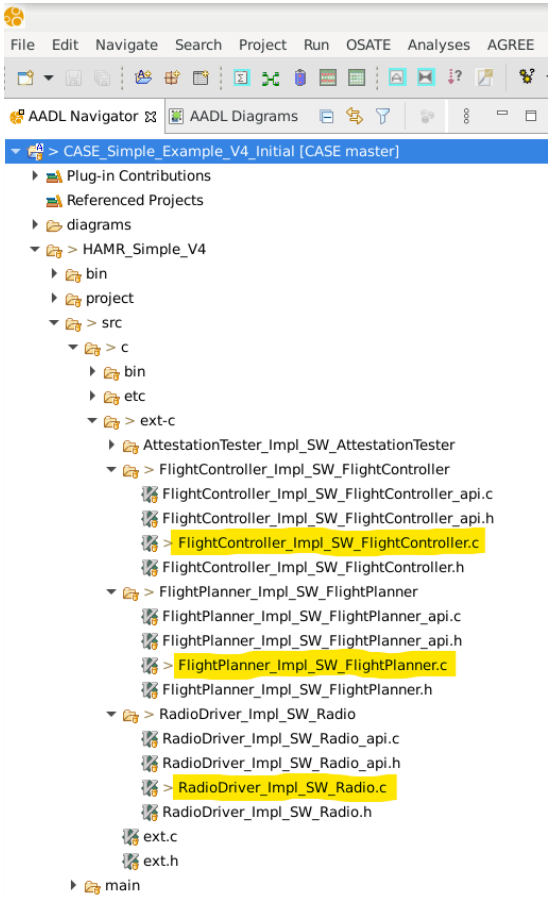


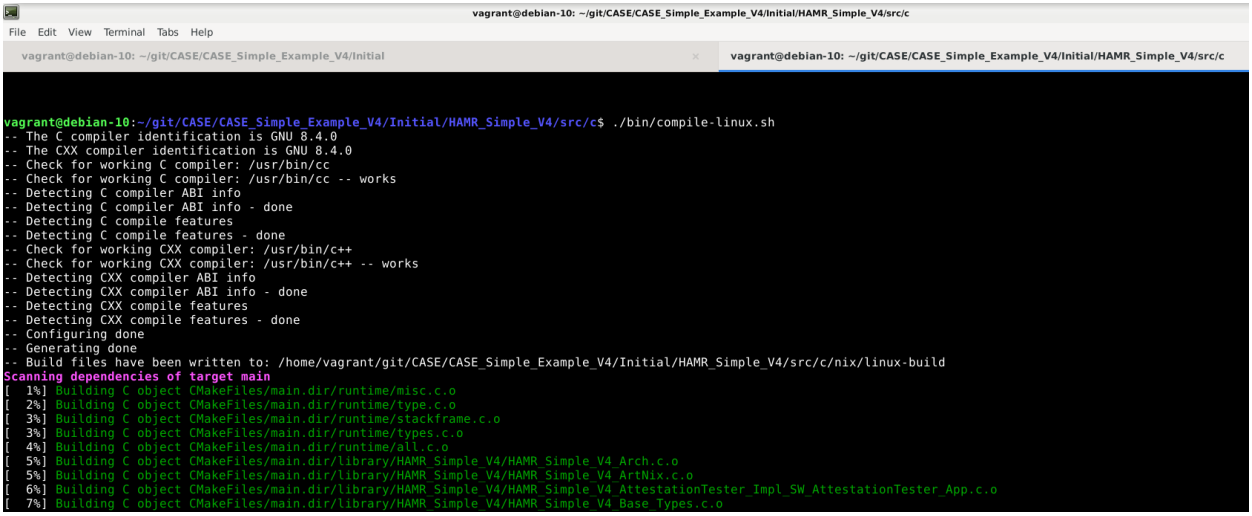
Figure 7. Directory structure for source code.

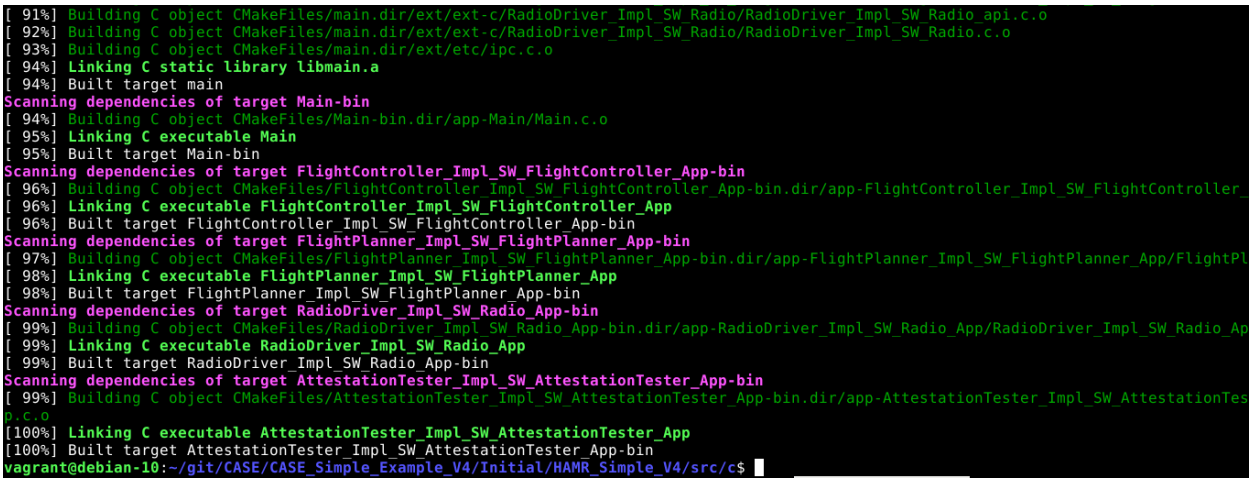
The files ext.h and ext.c are used to store commonly used routines and data in your component behavior code. If any new files are added to this structure, they will need to be included in HAMR's build procedure. We recommend modifying existing files to keep things simple.

We have provided C implementation files for the software components in the initial model and placed them in the Component_Source/Initial/ directory of the UAV project. These files are meant to replace the files generated by HAMR (assuming the naming convention used in this example is followed). It might be useful to compare these files with the ones generated by HAMR to understand (simple) usage of the HAMR API. Please note that when HAMR is run again, these implementation files will not be altered (which may or may not suit your purpose).

Once the component behaviors have been added, we compile them within the HAMR framework as shown below:

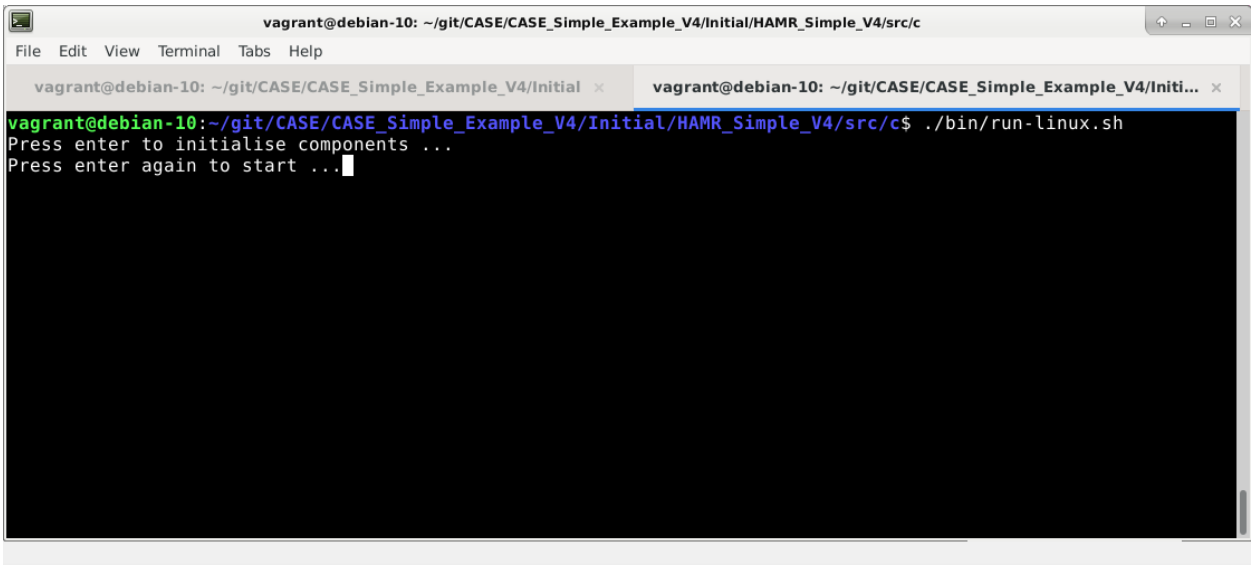
```
./bin/compile-linux.sh
```



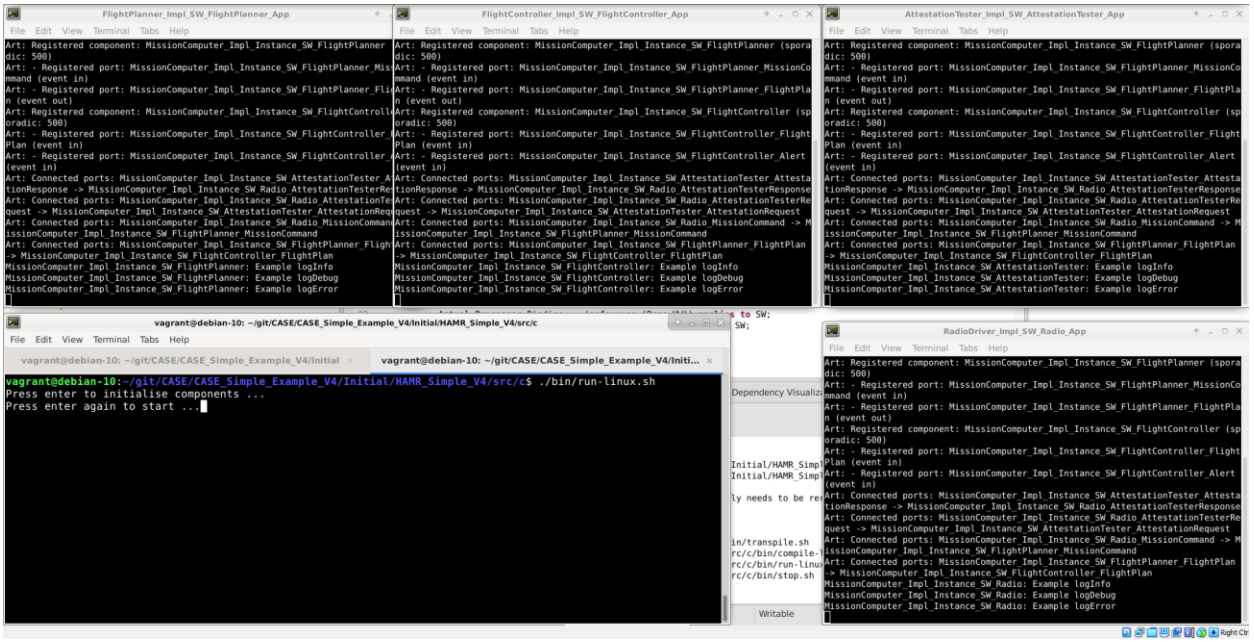


Finally, we run these components with:

```
./bin/run-linux.sh
```



The script brings up a terminal for each of the three executing components (showing their associated log outputs).



The execution is stopped by entering `./bin/stop.sh` in the console:

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c$ ./bin/stop.sh
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Initial/HAMR_Simple_V4/src/c$
```

4 Cyber Requirements

For this example, it is assumed that a cyber-requirements tool such as GearCASE was already run on the Mission Computer implementation (MC::MissionComputer.Impl), and that its output (GearCASE_Cyber_Requirements.json) is placed in the project's *Requirements* folder.

4.1 Importing Requirements

To import the generated requirements into the model so they can be addressed, open the MC.aadl file, select MissionComputer.Impl in the outline pane and choose BriefCASE → Cyber Requirements → Import Requirements… from the main menu. A file selection dialog will open. Navigate to the CASE_Simple_UAV_Example_v4/Requirements/ directory, select the GearCASE_Cyber_Requirements.json file, and click Open. The Requirements Import Manager will then open, displaying a list of generated requirements (see Figure 8).

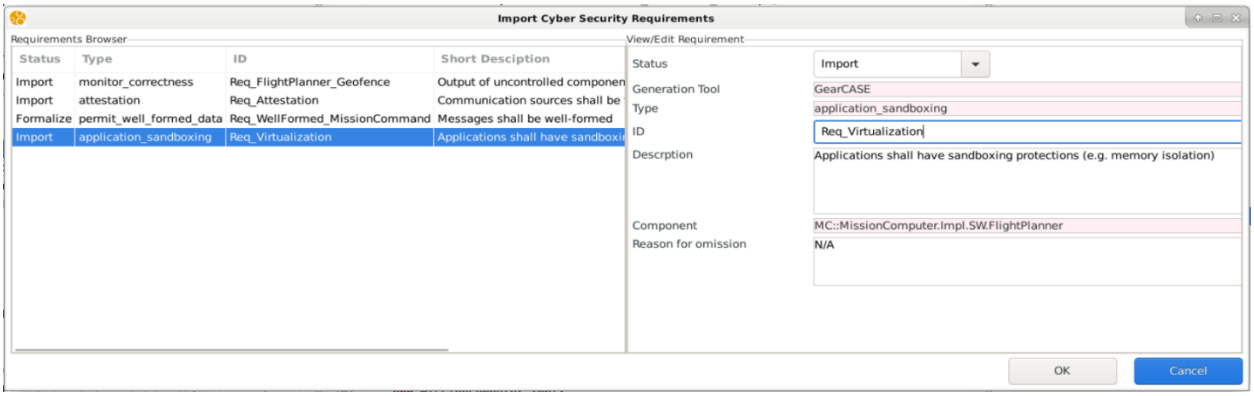


Figure 8. BriefCASE Requirements Import Manager.

For this example, four requirements were generated. For each requirement, set the Status and ID fields as shown in the figure and select OK. The requirements will then be imported into the model as Resolute goals and can be found in the Requirements/CASE_Requirements.aadl file.

Because the *Req_Wellformed_MissionCommand* requirement was set as formalized, an AGREE *assume* statement was generated for the FlightPlanner. By default, the formal expression is set to `false`, which will cause AGREE to fail if it is run. Modify the expression as shown below, which provides a true formalization of the requirement:

```
assume Req_WellFormed_MissionCommand "Messages shall be well-formed" :
      event(MissionCommand) => WELL_FORMED_MISSION_COMMAND(MissionCommand);
```

4.2 Requirements as Resolute Goals

The requirements are maintained as goals in a Resolute assurance case. Initially, the goals do not specify strategies or evidence to support the goals, so running Resolute at this time will generate a failing assurance case, as expected. To verify this, open MC.aadl in the editor, select MissionComputer.Impl in the Outline pane and choose Analyses → Resolute from the menu. The failing assurance case will appear with red exclamation marks in the Assurance Case pane at the bottom of the IDE, as shown in Figure 9.

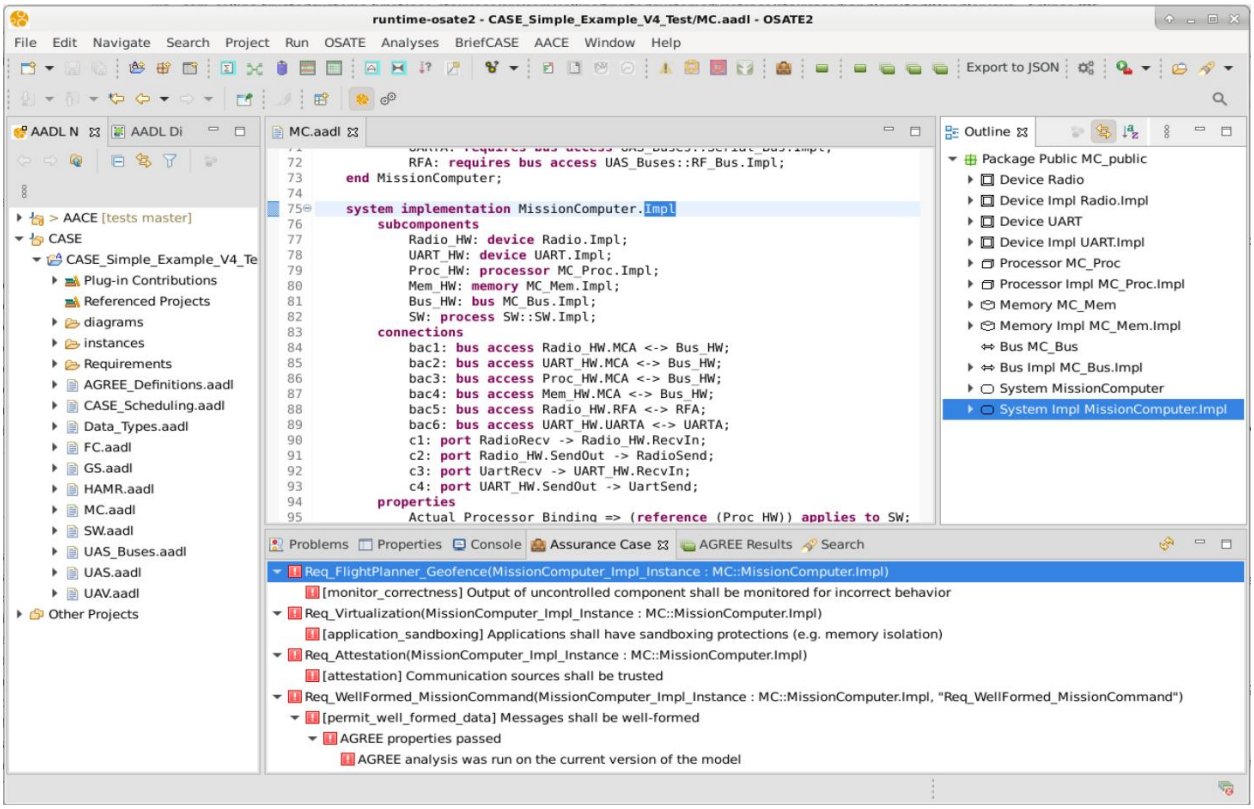


Figure 9. Resolute results.

5 Cyber-Resiliency Transforms

In this section we show how to use the cyber-resiliency transforms implemented in the BriefCASE tool. These transforms will address the cyber-requirements that have been imported into the model. The BriefCASE User's Guide contains additional information on how to perform each transformation, and refers to micro-examples included with the CASE virtual machine in the ~/CASE/transform-examples directory.

5.1 seL4 Transform

Because the target platform will be running seL4, the model can be transformed to reflect that runtime architecture. Since seL4 provides both memory (spatial) and execution (temporal) separation guarantees, the proper way to represent this in AADL is to have each thread run in its own dedicated process. When using BriefCASE, the model can be transformed for seL4 anytime during development. After the seL4 transformation has been performed, successive BriefCASE transformations give the user the option to generate seL4-formatted components.

To perform the seL4 transform, open the MC.aadl file in the editor and select the SW process subcomponent within the MissionComputer implementation, and choose BriefCASE → Cyber Resiliency → Model Transformations → Transform for seL4 Build… from the menu.

Once the transform has completed, a notification will appear in the lower right-hand corner of the screen, and both the MC.aadl and SW.aadl files will have been modified. The most obvious changes will

be in the SW.aadl file where, for each thread, a process is created containing that thread as a subcomponent. The SW component is also transformed from a process to a system.

If you wish to run this initial version of the system on seL4 (prior to applying the cyber-resiliency transforms), you can do so following the instructions in Section 8.

5.2 Attestation Transform

The *Req_Attestation* requirement can be mitigated by performing the Attestation transform. In SW.aadl, select the Radio_SeL4 process subcomponent within the SW_seL4 implementation, and choose BriefCASE → Cyber Resiliency → Model Transformations → Add Attestation… from the menu. A wizard will open, as shown in Figure 10.

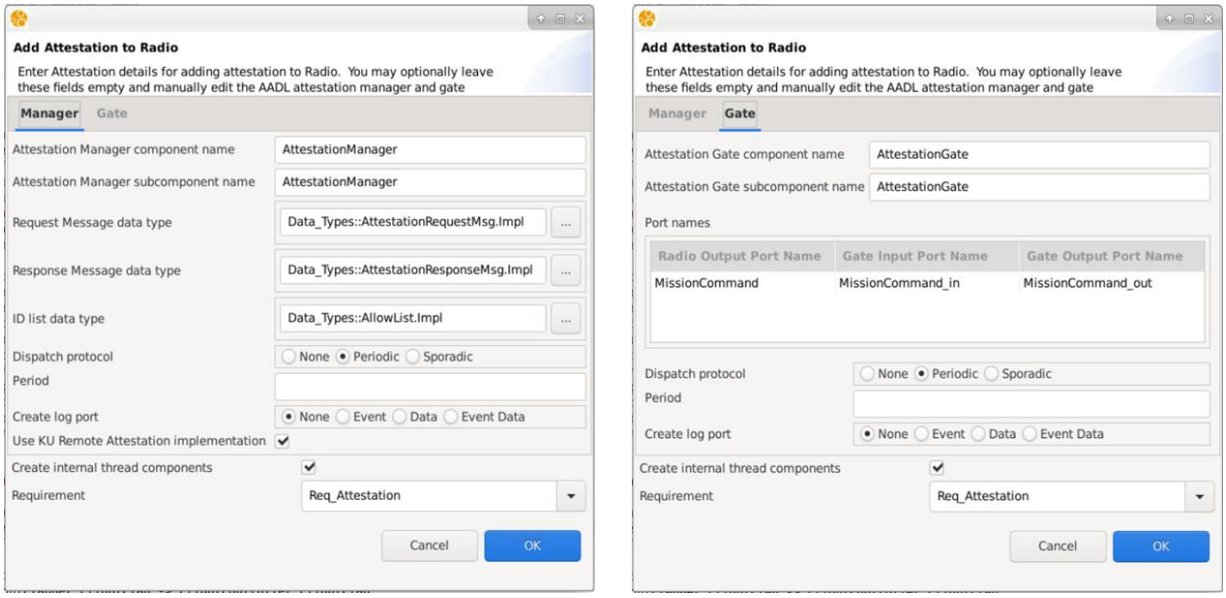


Figure 10. Attestation Transform wizard.

Note that the wizard contains two tabs, one for configuring the attestation manager, and the other for configuring the attestation gate. Fill in the fields on each as shown in the figure, then click OK. Attestation components will be inserted into the model, as shown in green in Figure 11.

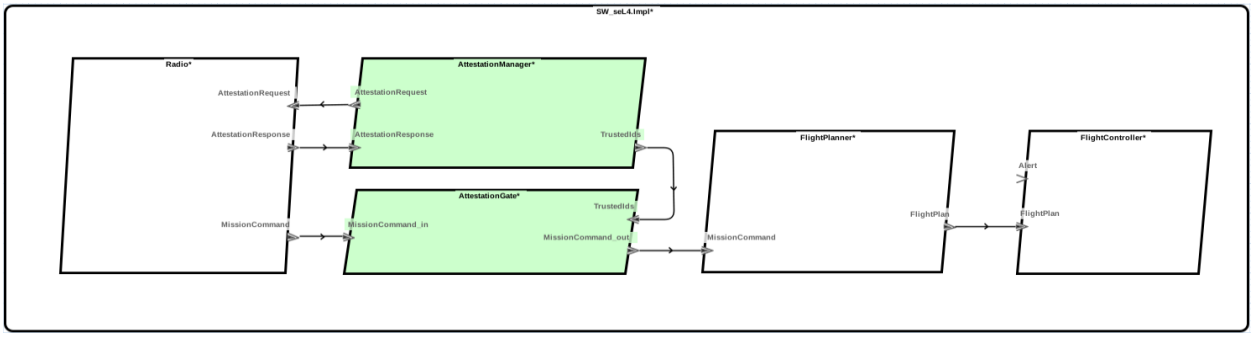


Figure 11. Attestation components inserted into model.

Although the components were automatically added to the model, the behavioral specification for the attestation gate must be entered manually. We plan to automate this in a future version of the tool. Copy the following AGREE annex to the CASE_AttestationGate thread:

```
annex agree {**

        eq trusted_ids : Data_Types::AllowList.Impl =
                (if event(TrustedIds) then TrustedIds else default_trustedid_list)
                ->
                (if event(TrustedIds) then TrustedIds else pre(trusted_ids));
        fun IS_TRUSTED(command : Data_Types::RF_Msg.Impl, ids : Data_Types::AllowList.Impl) :
        bool =
                exists i in ids.value, command.header.src = i;
        guarantee MissionCommand_Output "The gate shall output only data from trusted sources" :
                if event(MissionCommand_in) and IS_TRUSTED(MissionCommand_in, trusted_ids) then
                        event(MissionCommand_out) and MissionCommand_out = MissionCommand_in
                else
                        not event(MissionCommand_out);

**};
```

Also, add the following property to the CASE_AttestationGate thread:

```
CASE_Properties::Component_Spec => ("MissionCommand_Output");
```

When these additions have been made, the CASE_AttestationGate thread should appear as in Figure 12. Be sure to add these specifications to the CASE_AttestationGate *thread* component, and not the CASE_AttestationGate_seL4 *process*.

```
258    thread AttestationGate
259        features
260            MissionCommand_in: in event data port Data_Types::RF_Msg.Impl;
261            MissionCommand_out: out event data port Data_Types::RF_Msg.Impl;
262            TrustedIds: in event data port Data_Types::AllowList.Impl;
263        properties
264            CASE_Properties::Gating => 100;
265            CASE_Properties::Component_Spec => ("MissionCommand_Output");
266        annex agree {**
267            eq trusted_ids : Data_Types::AllowList.Impl =
268                (if event(TrustedIds) then TrustedIds else default_trustedid_list)
269                ->
270                (if event(TrustedIds) then TrustedIds else pre(trusted_ids));
271            fun IS_TRUSTED(command : Data_Types::RF_Msg.Impl, ids : Data_Types::AllowList.Impl) :
272                bool = exists i in ids.value, command.header.src = i;
273            guarantee MissionCommand_Output "The gate shall output only data from trusted sources" :
274                if event(MissionCommand_in) and IS_TRUSTED(MissionCommand_in, trusted_ids) then
275                    event(MissionCommand_out) and MissionCommand_out = MissionCommand_in
276                else
277                    not event(MissionCommand_out);
278        **};
279
280    end AttestationGate;
```

Figure 12. Attestation Gate specification in AGREE.

Because the KU implementation option was selected in the configuration wizard, the Attestation Manager implementation source code is added to the project directory, as shown in Figure 13.

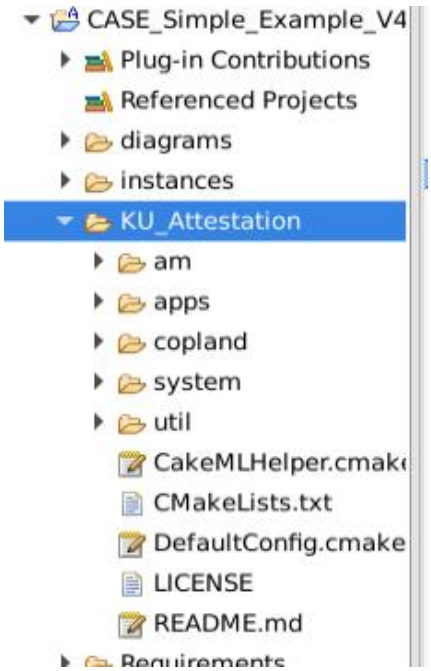


Figure 13. KU remote attestation implementation directory.

The transform also updated the *Req_Attestation* requirement in the CASE_Requirements.aadl file, as can be seen in Figure 14.

```
goal Req_Attestation(comp_context : component, attestation_manager : component, attestation_gate : component) <=
    ** "[attestation] Communication sources shall be trusted" **
    context Generated_By : "GearCASE";
    context Generated_On : "Aug 7, 2020";
    context Req_Component : "MC::MissionComputer.Impl.SW.Radio";
    context Formalized : "False";
    add_attestation_manager(comp_context, attestation_manager, attestation_gate)
```

Figure 14. Updated attestation requirement.

Because the requirement is being addressed by adding attestation components, a new rule is added to the Resolute goal that specifies the evidence required to show the goal has been satisfied.

The attestation gate implementation is synthesized by SPLAT, which will be described later in this document.

5.3 Filter Transform

To ensure messages received by the FlightPlanner are well-formed, a filter can be inserted immediately before it on connection c1, thereby addressing the *Req_Wellformed_MissionCommand* requirement. To insert the filter, select connection c1 within the SW system implementation either in the editor or the Outline pane, and choose BriefCASE → Cyber Resiliency → Model Transformations → Add Filter… from the menu. A configuration wizard will open, as shown in Figure 15.

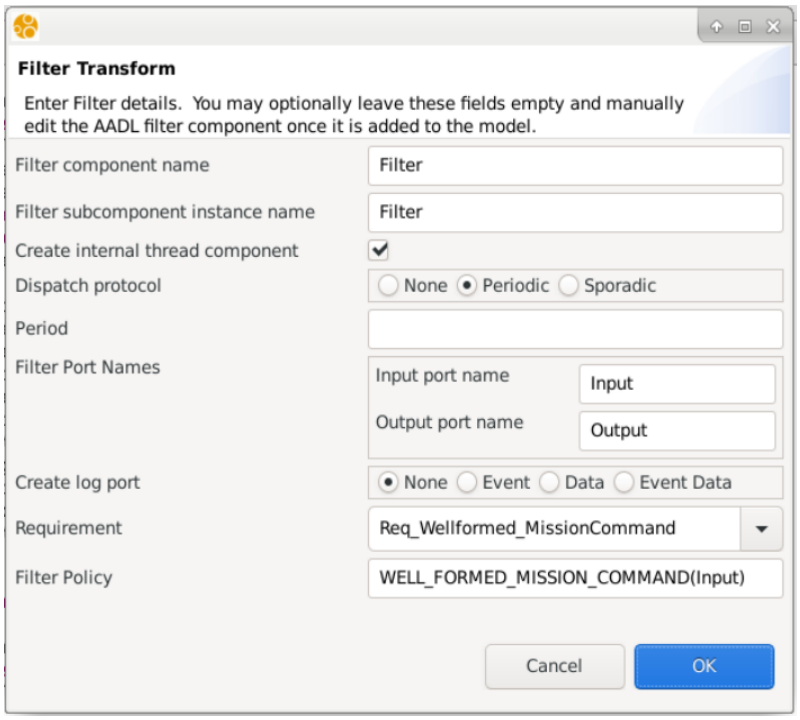


Figure 15. Filter Transform wizard.

Fill in the fields as shown in the figure, then click OK. For the filter policy, enter:

```
WELL_FORMED_MISSION_COMMAND(Input)
```

The details of this policy are already specified in the file AGREE_Definitions.aadl. It specifies that all of the waypoints must be valid latitude and longitude values.

A filter component will be inserted into the model, as shown in green in Figure 16.

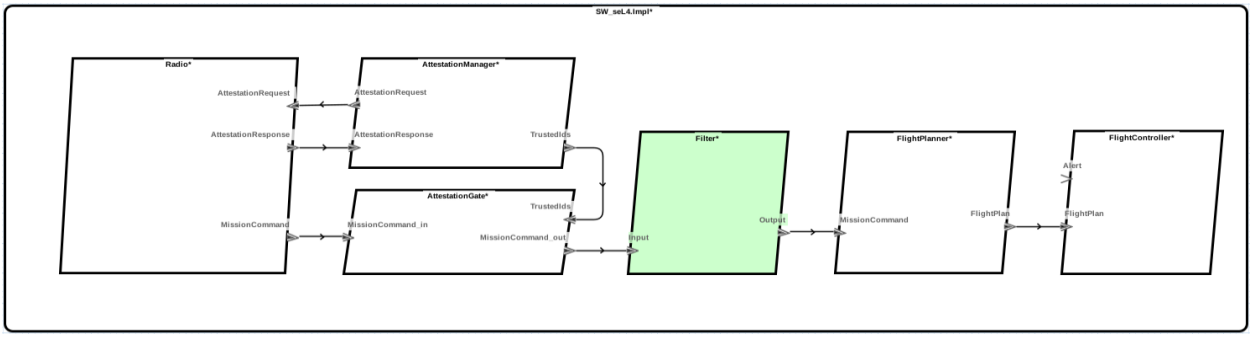


Figure 16. Filter component inserted into model.

The transform updated the *Req_Wellformed_MissionCommand* requirement in the CASE_Requirements.aadl file, as can be seen in Figure 17.

```
29⊖         goal Req_WellFormed_MissionCommand(comp_context : component, property_id : string, filter : component, conn : connection, message_type : data) <=
30              ** "[permit_well_formed_data] Messages shall be well-formed" **
31              context Generated_By : "GearCASE";
32              context Generated_On : "Aug 7, 2020";
33              context Req_Component : "MC::MissionComputer.Impl.SW.FlightPlanner";
34              context Formalized : "True";
35              agree_property_checked(comp_context, property_id) and add_filter(comp_context, filter, conn, message_type)
```

Figure 17. Updated filter requirement.

Two clauses in the Resolute goal check that (1) AGREE was run on the current version of the model (and passed), since this requirement is formalized, and (2) that the filter was inserted properly in the model and the implementation is correct.

The filter implementation is synthesized by SPLAT, which will be described later in this document.

5.4 Monitor Transform

Because the FlightPlanner component is considered *untrusted*, the *Req_FlightPlanner_Geofence* requirement was emitted to protect against suspicious behavior. To insert a monitor, select connection c2 within the SW system implementation either in the editor or the Outline pane, and choose BriefCASE → Cyber Resiliency → Model Transformations → Add Monitor… from the menu. A configuration wizard will open, as shown in Figure 18.

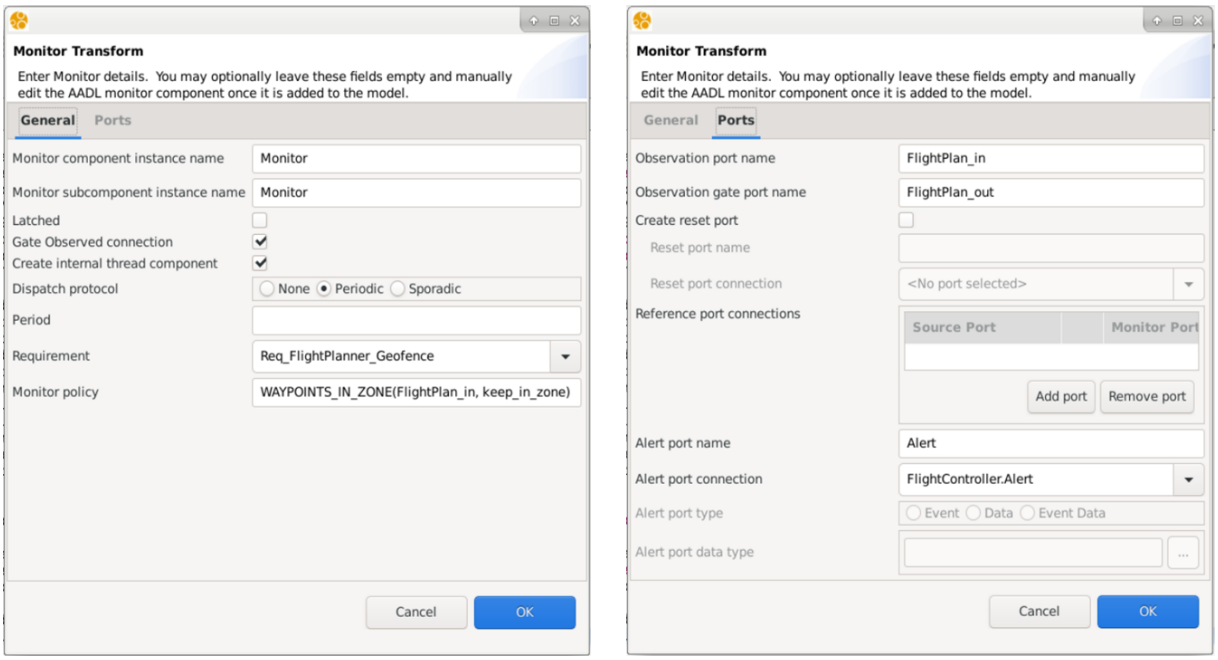


Figure 18. Monitor Transform wizard.

Note that the wizard contains two tabs. Fill in the fields on each tab as shown in the figure, then click OK. For the Monitor Policy, enter:

```
WAYPOINTS_IN_ZONE(FlightPlan_in, keep_in_zone)
```

The details of this policy are also specified in the file `AGREE_Definitions.aadl`. It checks that all of the waypoints lie within a pre-defined rectangular keep-in zone.

A monitor component will be inserted into the model, as shown in green in Figure 19.

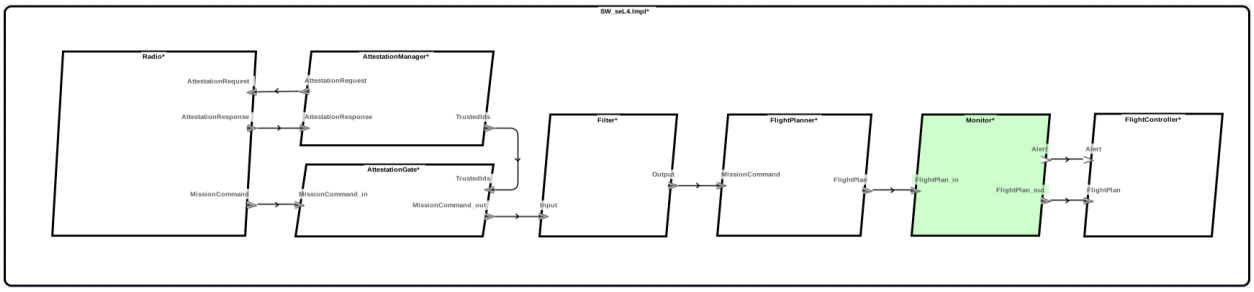


Figure 19. Monitor component inserted into model.

The transform updated the *Req_FlightPlanner_Geofence* requirement in the CASE_Requirements.aadl file, as can be seen in Figure 20.

```
29⊖     goal Req_FlightPlanner_Geofence(comp_context : component, monitor : component, alert_port : feature, gate_context : component, message_type : data) <=
30          ** "[monitor_correctness] Output of uncontrolled component shall be monitored for incorrect behavior" **
31          context Generated_By : "GearCASE";
32          context Generated_On : "Aug 7, 2020";
33          context Req_Component : "MC::MissionComputer.Impl.SW.FlightPlanner";
34          context Formalized : "False";
35          add_monitor_gate(comp_context, monitor, alert_port, gate_context, message_type)
```

Figure 20. Updated monitor requirement.

The monitor implementation is synthesized by SPLAT, which will be described later in this document.

5.5 Virtualization Transform

Like the monitor, the final *Req_Virtualization* requirement was generated because the FlightPlanner component is *untrusted*. More specifically, we suppose that it has some characteristics that require it to run on a guest OS in a virtual machine, placing it in its own isolated environment or *sandbox*. This may be because it requires some Linux devices or system capabilities not available in seL4, or because it would be too expensive to port the software to run on seL4.

We can sandbox the FlightPlanner in a virtual machine using the virtualization transform. To do so, select the SW system subcomponent in the MissionComputer.Impl in the MC.aadl file, and choose BriefCASE → Cyber Resiliency → Model Transformations → Add Virtualization… from the menu. A wizard will open, as shown in Figure 21.

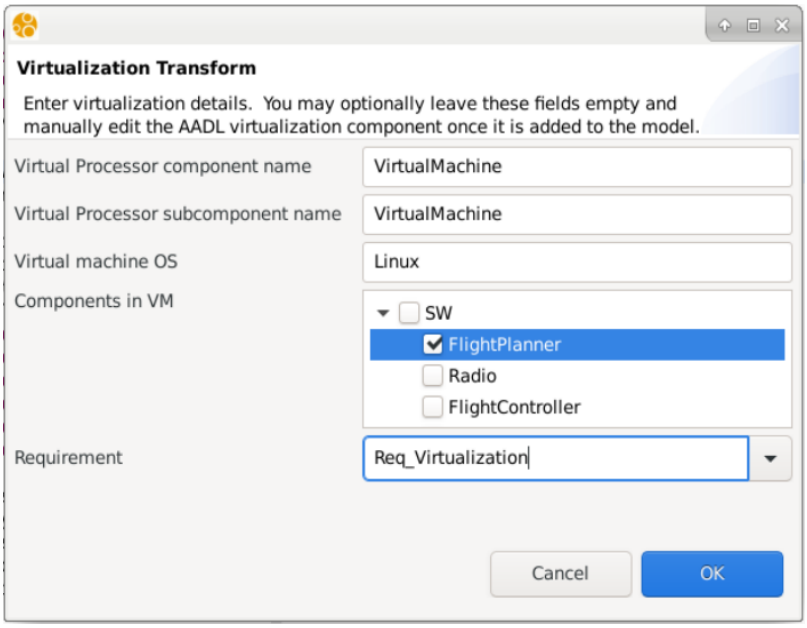


Figure 21. Virtualization Transform wizard.

Fill in the fields on each as shown in the figure, then click OK. The Virtualization transform inserts a virtual machine into the MissionComputer, and binds the FlightPlanner process to it.

For HAMR to generate the correct infrastructure code for the virtual machine, the following property needs to be manually added (it will be automatically inserted in future versions of the tool) to FlightPlanner_seL4.Impl:

```
HAMR::Component_Type => VIRTUAL_MACHINE;
```

However, we are not actually going to add this property yet. To keep things simple, the first version of the system we build will not include virtualization. We will return to this later, add the VIRTUAL_MACHINE property, and rebuild the final version of the system.

The transform updated the *Req_Virtualization* requirement, as shown in Figure 22.

```
29⊖     goal Req_Virtualization(comp_context : component, vm_components : {component}, virtual_machine : component) <=
30          ** "[application_sandboxing] Applications shall have sandboxing protections (e.g. memory isolation)" **
31          context Generated_By : "GearCASE";
32          context Generated_On : "Aug 7, 2020";
33          context Req_Component : "MC::MissionComputer.Impl.SW.FlightPlanner";
34          context Formalized : "False";
35          add_virtualization(vm_components, virtual_machine)
```

Figure 22. Updated virtualization requirement.

5.6 High-Assurance Component Synthesis

High-assurance components that have their behavior specified in AGREE can be synthesized using the SPLAT tool. To run SPLAT on the SW implementation, open the SW.aadl in the text editor, and select CASE → Cyber Resiliency → Synthesis Tools → SPLAT from the main menu, as shown in Figure 23.

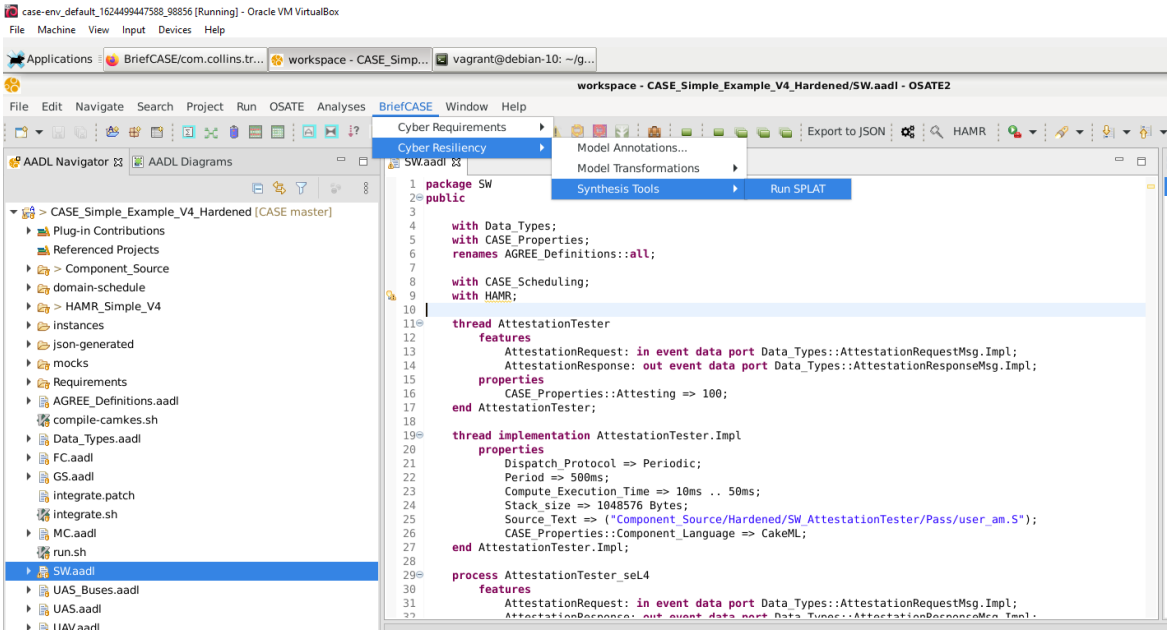


Figure 23. Running SPLAT.

As SPLAT runs, diagnostic messages will appear in the console at the bottom of the IDE, and a notification will be displayed in the lower right-hand corner when it completes. A maximized view of the console is shown in Figure 24.

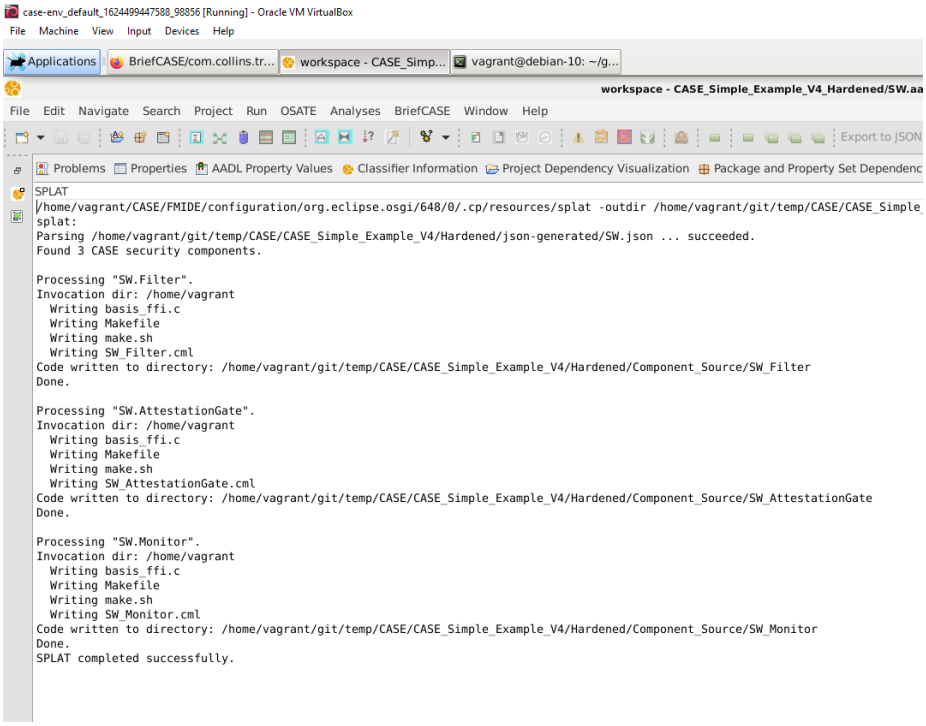


Figure 24. SPLAT console.

SPLAT outputs CakeML component implementations to the Component_Source folder, with each component implementation in a separate folder. In addition, the Source_Text property of each high-

assurance thread implementation will be set to the location of the corresponding source file in the project directory.

6 Analysis of the Cyber-Resilient System

Now that we have transformed the system to address the cyber-requirements, we can analyze the resulting model using AGREE (formal verification of behaviors) and Resolute (generation and checking of an assurance case).

6.1 Formal Verification using AGREE

Although formal verification of the model is not the focus of this example, AGREE can still be run on the model to verify that it satisfies its contracts. To run AGREE, select the SW system implementation in SW.aadl, then choose Analyses → AGREE → Verify Single Layer from the menu. The results will display in the AGREE results pane, and should pass, as indicated by green checkmarks (see Figure 25).

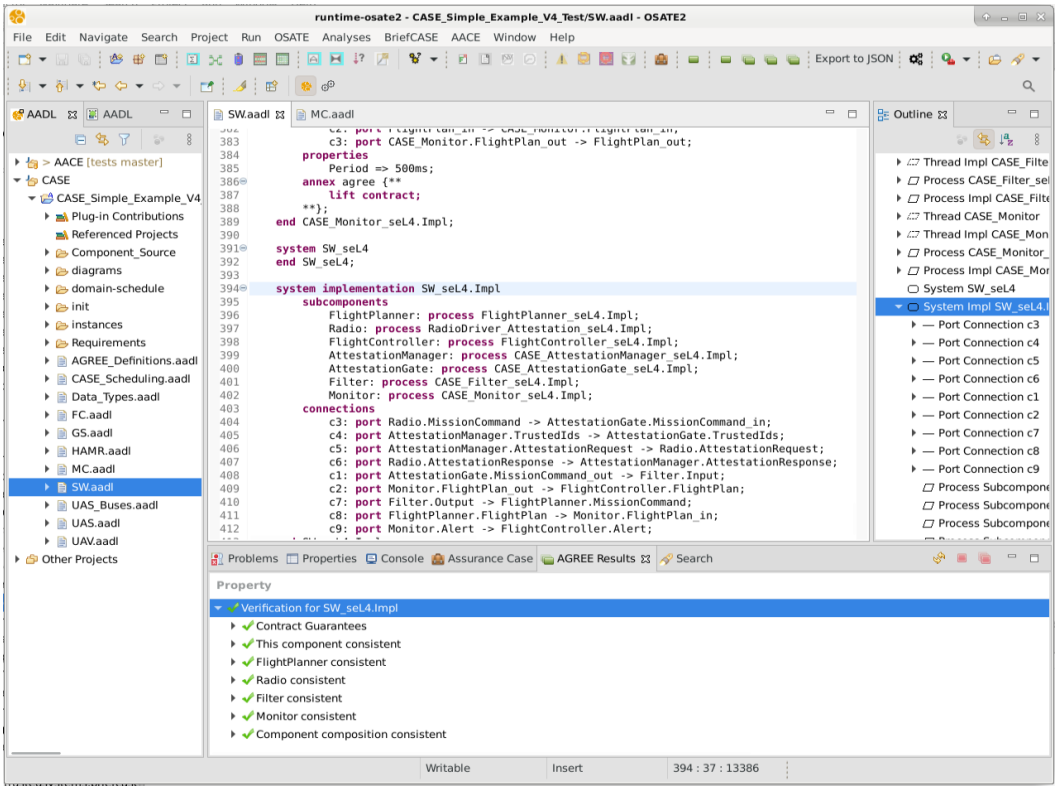


Figure 25. AGREE formal verification results.

6.2 Cyber Assurance Case

Now that all requirements have been addressed, high-assurance components synthesized, and formal verification performed, Resolute should produce a passing assurance case. To confirm this, select the MissionComputer implementation in MC.aadl, and choose Analyses → Resolute from the menu. The results will appear in the Assurance Case pane.

7 Preparing to Build the Cyber-Resilient System

There are several additions to the model that should be performed next to help build the final cyber-resilient system.

7.1 Adding an Attestation Test Harness

The Radio component represents a communication driver that receives messages from a remote system. However, for simplicity, the actual implementation included with this example generates the command message itself and places it on the MissionCommand port. This poses a problem for building and testing a system that employs remote attestation, since attestation request and response messages are passed between attestation components on two communicating systems.

We therefore include an attestation test harness with the example. It can be connected to the radio driver and behaves as if it is running on a remote system. The SW.aadl package already includes an AttestationTester component; it just needs to be properly configured and wired into the SW system.

Two different attestation implementations are included with the harness: one that will produce a "good" result, representing an uncompromised remote system, and the other producing a "bad" result, indicating that the remote system has been compromised. The supplied version of the AttestationTester is configured to provide a passing measurement, as shown in Figure 26. To provide a failing measurement, simply change "Pass" to "Fail" in the `Source_Text` property path on line 25, as shown in Figure 27.

```
11  thread AttestationTester
12      features
13          AttestationRequest: in event data port Data_Types::AttestationRequestMsg.Impl;
14          AttestationResponse: out event data port Data_Types::AttestationResponseMsg.Impl;
15      properties
16          CASE_Properties::Attesting => 100;
17  end AttestationTester;
18
19  thread implementation AttestationTester.Impl
20      properties
21          Dispatch_Protocol => Periodic;
22          Period => 500ms;
23          Compute_Execution_Time => 10ms .. 50ms;
24          Stack_size => 1048576 Bytes;
25          Source_Text => ("Component_Source/Hardened/SW_AttestationTester/Pass/user_am.S");
26          CASE_Properties::Component_Language => CakeML;
27  end AttestationTester.Impl;
```

Figure 26. Attestation Tester implementation for producing a good result.

```
11  thread AttestationTester
12      features
13          AttestationRequest: in event data port Data_Types::AttestationRequestMsg.Impl;
14          AttestationResponse: out event data port Data_Types::AttestationResponseMsg.Impl;
15      properties
16          CASE_Properties::Attesting => 100;
17  end AttestationTester;
18
19  thread implementation AttestationTester.Impl
20      properties
21          Dispatch_Protocol => Periodic;
22          Period => 500ms;
23          Compute_Execution_Time => 10ms .. 50ms;
24          Stack_size => 1048576 Bytes;
25          Source_Text => ("Component_Source/Hardened/SW_AttestationTester/Fail/user_am.S");
26          CASE_Properties::Component_Language => CakeML;
27  end AttestationTester.Impl;
```

Figure 27. Attestation Tester implementation for producing a failing result.

Now we will connect the AttestationTester to the Radio component. This requires making modifications to the SW_sel4 and RadioDriver_Attestation_seL4 process definitions, as well as the RadioDriver_Attestation thread definition. Modify the RadioDriver_Attestation thread to appear as shown in Figure 28, and the corresponding RadioDriver_Attestation_seL4 process type and implementation to appear as shown in Figure 29. These modifications add the ports to the Radio component for communicating with the AttestationTester.

```
thread RadioDriver_Attestation
    features
        MissionCommand: out event data port Data_Types::RF_Msg.Impl;
        AttestationRequest: in event data port Data_Types::AttestationRequestMsg.Impl;
        AttestationResponse: out event data port Data_Types::AttestationResponseMsg.Impl;
        AttestationTesterResponse: in event data port Data_Types::AttestationResponseMsg.Impl;
        AttestationTesterRequest: out event data port Data_Types::AttestationRequestMsg.Impl;
    properties
        CASE_Properties::Comm_Driver => true;
end RadioDriver_Attestation;
```

Figure 28. Modified RadioDriver_Attestation thread with ports for communication with the AttestationTester. Modifications are in red box.

```
process RadioDriver_Attestation_seL4
    features
        MissionCommand: out event data port Data_Types::RF_Msg.Impl;
        AttestationRequest: in event data port Data_Types::AttestationRequestMsg.Impl;
        AttestationResponse: out event data port Data_Types::AttestationResponseMsg.Impl;
        AttestationTesterResponse: in event data port Data_Types::AttestationResponseMsg.Impl;
        AttestationTesterRequest: out event data port Data_Types::AttestationRequestMsg.Impl;
    properties
        CASE_Properties::Comm_Driver => true;
end RadioDriver_Attestation_seL4;

process implementation RadioDriver_Attestation_seL4.Impl
    subcomponents
        RadioDriver: thread RadioDriver_Attestation.Impl;
    connections
        c1: port RadioDriver.AttestationResponse -> AttestationResponse;
        c2: port AttestationRequest -> RadioDriver.AttestationRequest;
        c3: port RadioDriver.MissionCommand -> MissionCommand;
        c4: port AttestationTesterResponse -> RadioDriver.AttestationTesterResponse;
        c5: port RadioDriver.AttestationTesterRequest -> AttestationTesterRequest;
    properties
        Period => 500ms;
end RadioDriver_Attestation_seL4.Impl;
```

Figure 29. Modified RadioDriver_Attestation_seL4 process with ports and connections for communication with the AttestationTester. Modifications are in red boxes.

Next, add the AttestationTester subcomponent, along with new connections between the Radio and AttestationTester components by modifying the SW_seL4 system implementation to appear as shown in Figure 30.

```
system implementation SW_seL4.Impl
    subcomponents
        FlightPlanner: process FlightPlanner_seL4.Impl;
        Radio: process RadioDriver_Attestation_seL4.Impl;
        FlightController: process FlightController_seL4.Impl;
        AttestationManager: process CASE_AttestationManager_seL4.Impl;
        AttestationGate: process CASE_AttestationGate_seL4.Impl;
        Filter: process CASE_Filter_seL4.Impl;
        Monitor: process CASE_Monitor_seL4.Impl;
        AttestationTester: process AttestationTester_seL4.Impl;
    connections
        c3: port Radio.MissionCommand -> AttestationGate.MissionCommand_in;
        c4: port AttestationManager.TrustedIds -> AttestationGate.TrustedIds;
        c5: port AttestationManager.AttestationRequest -> Radio.AttestationRequest;
        c6: port Radio.AttestationResponse -> AttestationManager.AttestationResponse;
        c1: port AttestationGate.MissionCommand_out -> Filter.Input;
        c2: port Monitor.FlightPlan_out -> FlightController.FlightPlan;
        c7: port Filter.Output -> FlightPlanner.MissionCommand;
        c8: port FlightPlanner.FlightPlan -> Monitor.FlightPlan_in;
        c9: port Monitor.Alert -> FlightController.Alert;
        c10: port Radio.AttestationTesterRequest -> AttestationTester.AttestationRequest;
        c11: port AttestationTester.AttestationResponse -> Radio.AttestationTesterResponse;
end SW_seL4.Impl;
```

Figure 30. Modified SW_seL4 system with AttestationTester subcomponent and connections. Modifications are in red boxes.

7.2 Creating a Domain Schedule

The *AADL Modeling Guidelines for CASE* document included with BriefCASE provides detail on constructing a domain schedule as well as the AADL property associations necessary for HAMR to generate the schedule for execution on the seL4 platform. For this example, we provide the component scheduling property values necessary for the build to run on the target platform, but refer the reader to the Modeling Guidelines and related HAMR and seL4 documentation for a more comprehensive understanding of the process.

Periodic Dispatch: All threads must have the `Dispatch_Protocol` property set to `Periodic`. The threads in the initial version of the model already had this property set, and it was also set in the Attestation, Filter, and Monitor transformation wizards. However, if any other threads were added to the software, the thread implementation would need to add the property association as

```
Dispatch_Protocol => Periodic;
```

Domain: Each AADL process is associated with a unique numeric domain identifier specified using the `CASE_Scheduling::Domain` property. The domain numbering for processes starts at two, since domain 0 is reserved for seL4 operations, and domain 1 is reserved for the pacer. Assign domains to process component implementations using the `CASE_Scheduling::Domain` property association according to the following table:

Process	Domain
AttestationTester_seL4.Impl	2
RadioDriver_Attestation_seL4.Impl	3
AttestationManager_seL4.Impl	4
AttestationGate_seL4.Impl	5

Filter_seL4.Impl	6
FlightPlanner_seL4.Impl	7
Monitor_seL4.Impl	8
FlightController_seL4.Impl	9

Maximum Domain Value: The maximum value of the domain identifiers is specified in the AADL model using the `CASE_Scheduling::Max_Domain` property applied to the processor component that hosts the processes. Therefore, add the following property association to MissionComputer.Impl in MC.aadl:

```
CASE_Scheduling::Max_Domain => 10 applies to Proc_HW;
```

Compute Execution Time: The `Timing_Properties::Compute_Execution_Time` property is applied to each thread, and specifies the duration for which the thread is scheduled. This property is used by HAMR to generate a hint in the auto-generated schedule skeleton. The value should match the slot duration value length in the seL4 domain schedule (this needs to be checked manually). For each thread, add the following property association:

```
Timing_Properties::Compute_Execution_Time => 10ms..50ms;
```

Period: The `Timing_Properties::Period` property is applied to each thread, and specifies its period. Comments containing this information are included along with the HAMR generated skeleton for the schedule, helping the developer to verify that the periodicity of the thread implied by the written schedule matches the specified period in the AADL model (this needs to be checked manually). For each thread, add the following property association:

```
Timing_Properties::Period => 500 ms;
```

Clock Period: The tick duration for the underlying platform is configured using the `TimingProperties::Clock_Period`, applied to the processor component that hosts the processes. The slot durations within the schedule are specified in terms of ticks (in seL4, 1 tick = 2ms). Therefore, add the following property association to MissionComputer.Impl in MC.aadl:

```
TimingProperties::Clock_Period => 2ms applies to Proc_HW;
```

Frame Period: The period of the major frame of the schedule is specified using the `Timing_Properties::Frame_Period` property on the processor component in the AADL model. The sum of the domain lengths specified in the domain schedule should equal this value. The frame period value is included in hints generated by HAMR to accompany an auto-generated schedule skeleton. Correspondence to the actual duration of the schedule must be checked manually. Add the following property association to MissionComputer.Impl in MC.aadl:

```
TimingProperties::Frame_Period => 1000ms applies to Proc_HW;
```

Domain Schedule: A domain schedule for the example model is included in the project directory (/domain-schedule/schedule.c). The schedule itself is expressed as a C data structure used to configure seL4. The data structure is captured in its own source code file, and referenced within the model via the

`CASE_Scheduling::Schedule_Source_Text` property defined on the processor component. Therefore, add the following property association to MissionComputer.Impl in MC.aadl:

```
CASE_Scheduling::Schedule_Source_Text => "domain-schedule/schedule.c"
applies to Proc_HW;
```

7.3 Other AADL Properties

Additional properties should be set to enable HAMR code generation:

Stack Size: The maximum stack size requirements for each thread should be specified using the `Memory_Properties::Stack_Size` property. For this example, each thread can be assigned the same maximum stack size using the following property association:

```
Memory_Properties::Stack_Size => 1048576 Bytes;
```

Platform: The `HAMR::Platform` property defines for HAMR the target execution platform(s). Add the following property association to MissionComputer.Impl in MC.aadl:

```
HAMR::Platform => (seL4);
```

7.4 Checking Model Compliance with Style Guidelines

Before building the system, it is important to check that the model complies with the style guidelines. The CASE Modeling Guidelines are included with BriefCASE, and the rules have been formalized as Resolint statements. Running Resolint on the model will produce errors or warnings if the model is out of compliance with any of the guidelines. To run Resolint, a ruleset needs to be specified. The HAMR ruleset is specified in the Resolute annex of the MissionComputer implementation. To run Resolint on the example, select MissionComputer.Impl in MC.aadl, then choose Analyses → Resolint → Run Resolint from the menu. After Resolint runs, an information dialog will display the number of errors and warnings detected, each of which are listed in the Problems pane at the bottom of the IDE. Double-clicking on one of the problems will auto-navigate to the declaration of the AADL element that is violating the rule.

Once all Resolint problems have been addressed, the model is ready to be built using HAMR.

8 Building the Cyber-Resilient System

Now we are ready to build the hardened, cyber-resilient version of the simple UAV example system. The code will be compiled for seL4 running on the QEMU emulator. HAMR can work with both x86 and ARM architectures. The first version of the hardened system (without virtualization of the FlightPlanner) will be built for x86 (emulated in QEMU). Since the current version of the VM only supports ARM architecture, we will then build the hardened system with the VM for ARM (emulated in QEMU).

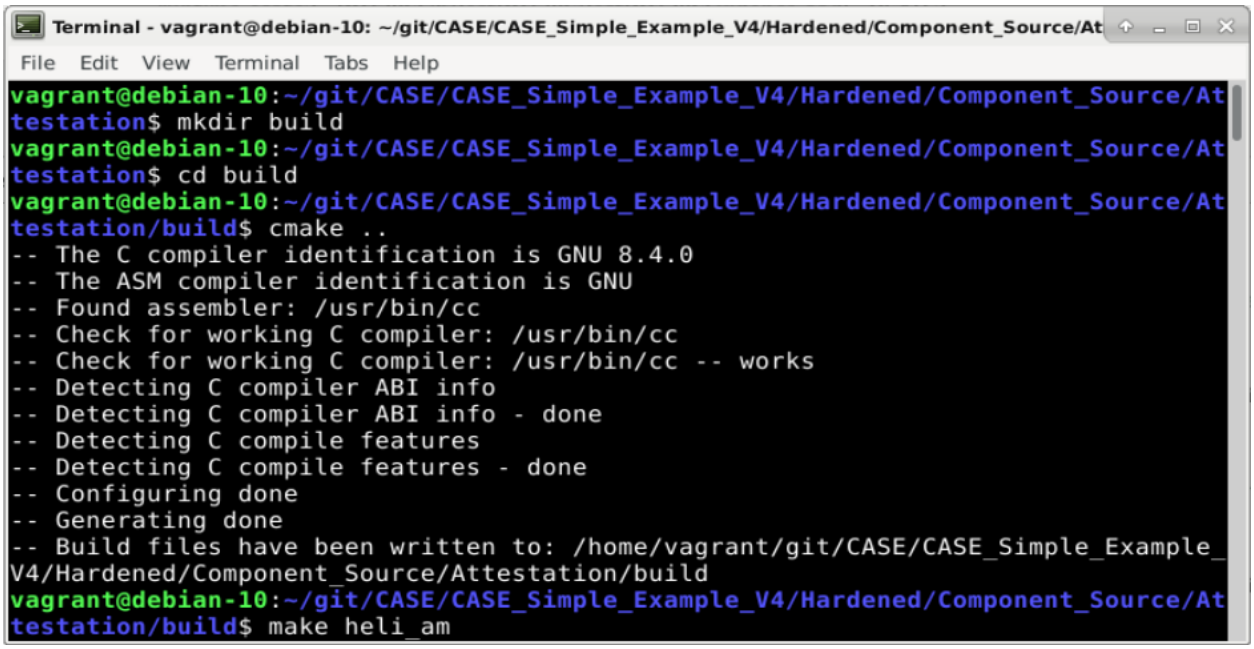
8.1 Compile SPLAT components

One last step in the synthesis of the high-assurance components is in building them. SPLAT places a Makefile in each of the synthesized component folders. It can be invoked as follows:

Note that we also copied SW_Filter.S to Filter.S since that is where BriefCASE expects to find the compiled high-assurance component (as annotated in SW.aadl). Also note that we must build each of the three high-assurance components similarly.

8.2 Compile the Attestation Manager

The KU source code for the Attestation Manager implementation was inserted into the Component_Source/Attestation directory at the time of the Attestation transform. It can be compiled as follows:



This will produce heli_am.S at the top-level of the build directory, and libheli_am_c.a in build/apps/case-tool-assessment.

8.3 Building the Hardened System (without VM component)

As mentioned previously, the first version of the system that we build will not include virtualization of the Flight Planner.

The first step is to generate the infrastructure code using HAMR. We run HAMR on the Mission Computer by selecting MissionComputer.Impl in the Outline pane (see Figure 3) and clicking the HAMR button in the OSATE toolbar.

In the HAMR dialog box that appears, select the platform as seL4 and specify the output directory, which we refer to as *HAMR_ROOT* from here on. For this example, the output directory must be named *HAMR_Simple_V4* to avoid linking errors with the included component implementations. For

convenience, we select the seL4/CAmkES output directory to be *HAMR_Simple_V4/CAmkES.* As before, we exclude slang components in the dialog box and select 64-bit width, as shown in Figure 31, before clicking Run.

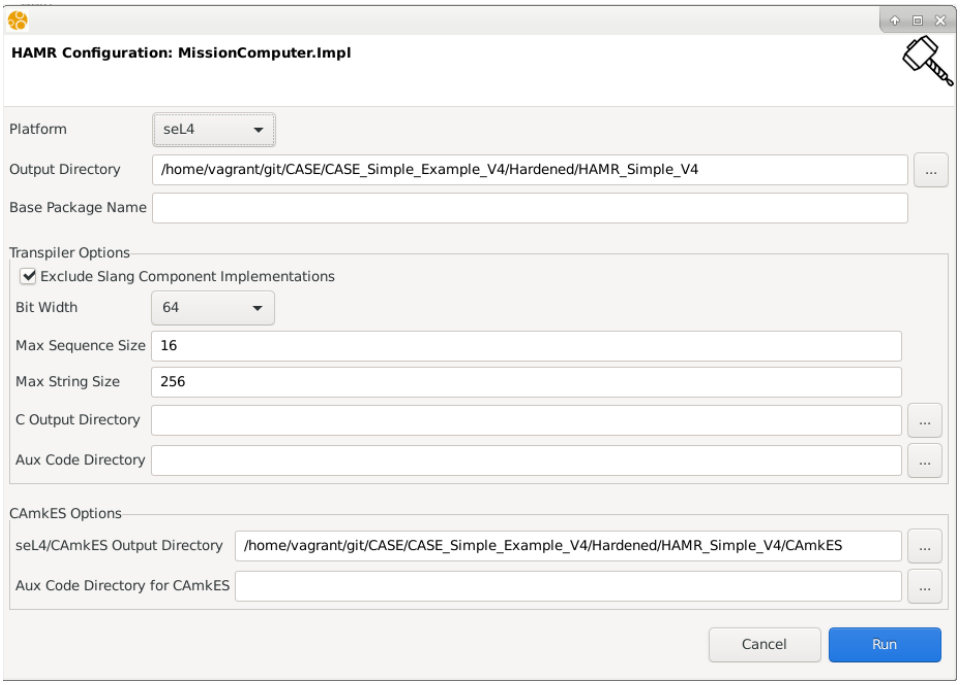
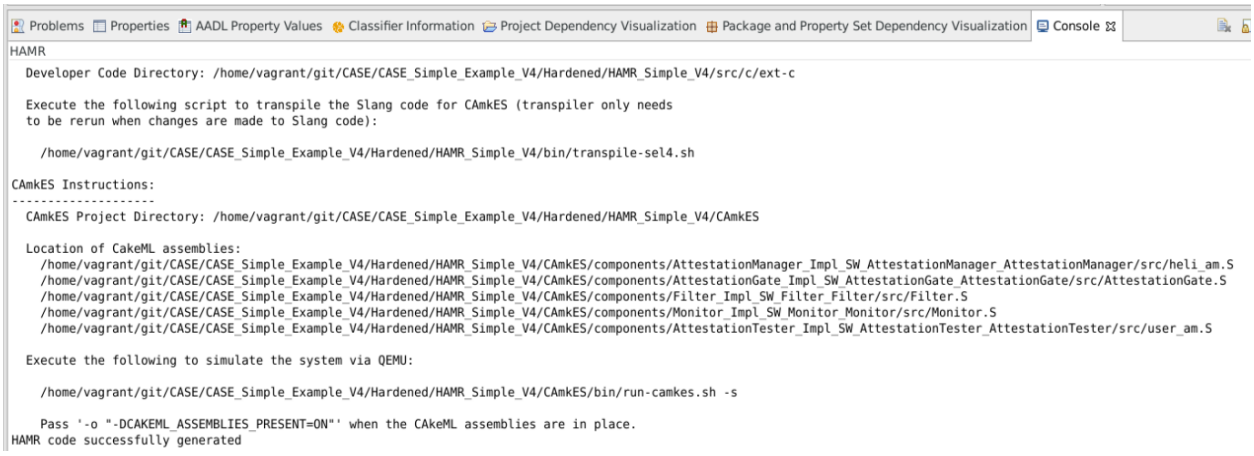


Figure 31. HAMR configuration dialog for hardened system.

On completion, HAMR's output should look like this:



Note that the instructions for building the hardened system are different from that of the initial model.

First, we run the transpiler as follows (this takes a much longer time for an seL4 build than for Linux):

```
./HAMR_Simple_V4/bin/transpile-sel4.sh
```

As shown in Figure 32, the directory structure generated by HAMR is quite different than for the initial model. The component implementations are spread across two folders within HAMR_Simlple_V4,

src/c/ext-c (for the C implementations) and CAmkES/components (for the CakeML/CAmkES components).

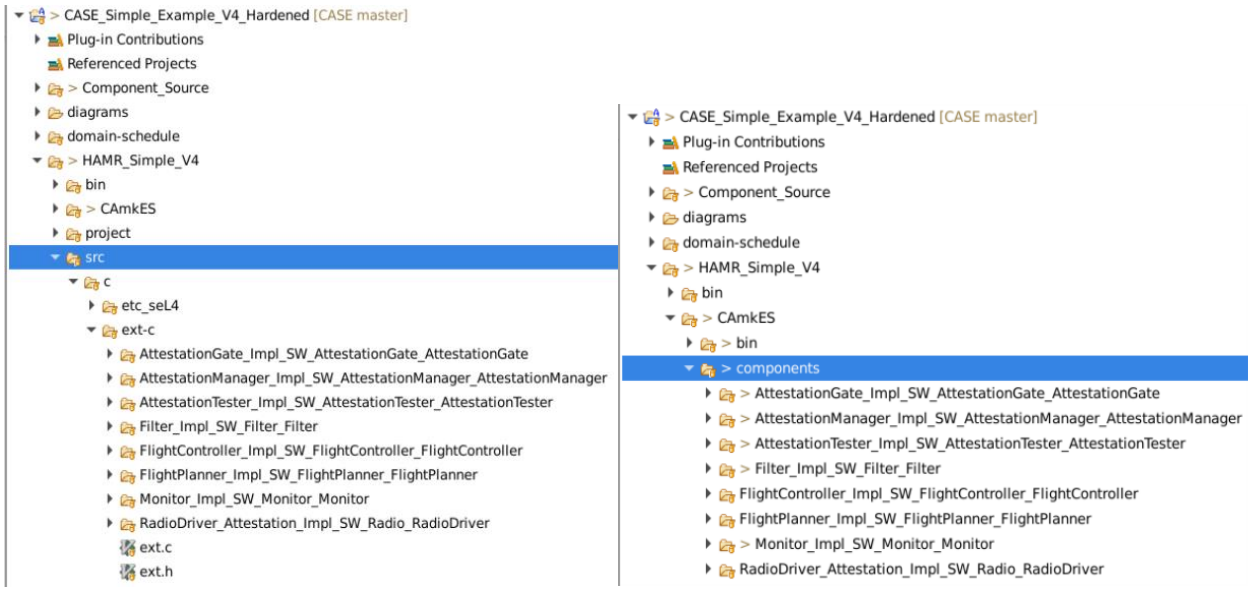
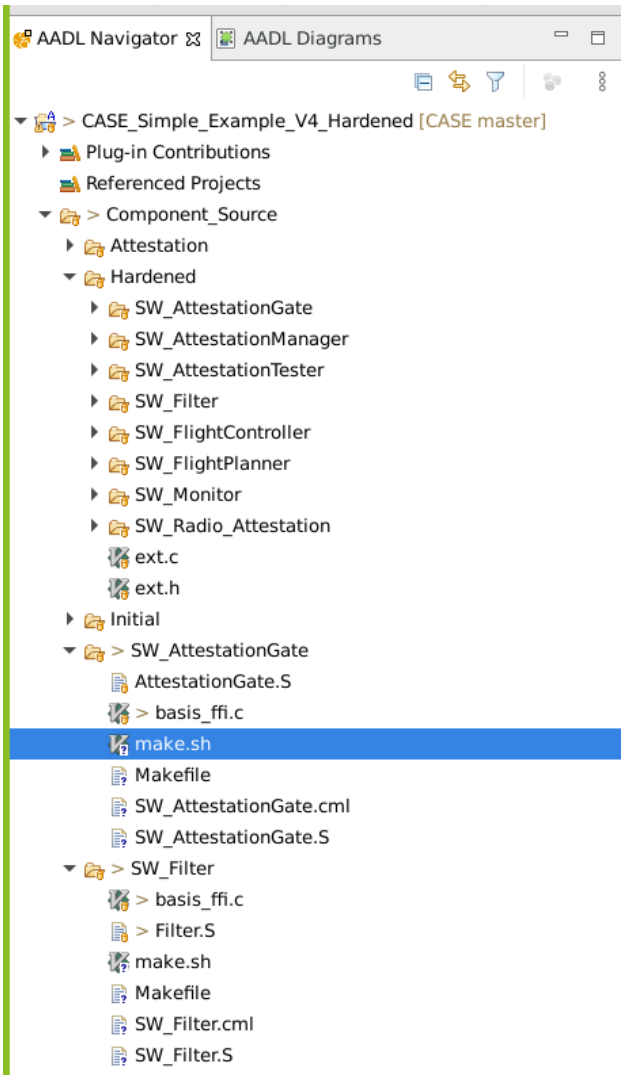


Figure 32. Location of component implementation code: C (left) and CakeML (right).

We have provided implementations for these components in the Component_Source folder as shown below:



For each of the CakeML components, we have also provided an equivalent hand-coded C implementation to support testing on Linux, if desired. For this example, we provide an *integrate.sh* script in the project root folder that moves the implementation files to their respective locations and modifies HAMR's compile scripts to include the copied files for compilation. Note that this is a custom file specific to this example and is provided to demonstrate the manual steps required for placing source code in the appropriate directories for the HAMR build and would need to be updated for your specific system and models. You could also perform these steps manually.

The script can be executed as follows:

```
./integrate.sh
```

After moving the files to the correct locations, *integrate.sh* also compiles them:

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened$ ./integrate.sh
~/git/CASE/CASE_Simple_Example_V4/Hardened/HAMR_Simple_V4 ~/git/CASE/CASE_Simple_Example_V4/Hardened
patching file ./CAmkES/CMakeLists.txt
patching file ./CAmkES/components/AttestationGate_Impl_SW_AttestationGate_AttestationGate/src/sb_AttestationGate_Impl_ffi.c
patching file ./CAmkES/components/Filter_Impl_SW_Filter_Filter/src/sb_Filter_Impl_ffi.c
patching file ./CAmkES/components/Monitor_Impl_SW_Monitor_Monitor/src/sb_Monitor_Impl_ffi.c
~/git/CASE/CASE_Simple_Example_V4/Hardened
'/home/vagrant/CASE/camkes/projects/camkes/apps/CAmkES' -> '/home/vagrant/git/CASE/CASE_Simple_Example_V4/Hardened/HAMR_Simple_V4/CAmkES'
Non-interactive mode so not deleting existing /home/vagrant/CASE/camkes/build_CAmkES
loading initial cache file /home/vagrant/CASE/camkes/projects/camkes/settings.cmake
-- Set platform details from PLATFORM=x86_64
--   KernelPlatform: pc99
--   KernelSel4Arch: x86_64
-- Found extra flag for SW_AttestationTester_AttestationTester.instance.bin: /ho
-- Found extra flag for SW_AttestationTester_AttestationTester.instance.bin:
-- CPIO test cpio_reproducible_flag PASSED
-- Configuring done
-- Generating done
-- Build files have been written to: /home/vagrant/CASE/camkes/build_CAmkES
[905/905] Generating images/capdl-loader-image-x86_64-pc99
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened$
```

Once compiled, the QEMU emulator can be invoked using *run.sh* as follows:

```
./run.sh
```

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened$ ./run.sh
'/home/vagrant/CASE/camkes/projects/camkes/apps/CAmkES' -> '/home/vagrant/git/CASE/CASE_Simple_Example_V4/Hardened/HAMR_Simple_V4/CAmkES'
Non-interactive mode so not deleting existing /home/vagrant/CASE/camkes/build_CAmkES
loading initial cache file /home/vagrant/CASE/camkes/projects/camkes/settings.cmake
-- Set platform details from PLATFORM=x86_64
--   KernelPlatform: pc99
--   KernelSel4Arch: x86_64
```

This script compiles any changes (if necessary), builds an image and loads it into the QEMU environment for simulation.

Building the image:

```
-- Found extra flags for SW_AttestationGate_AttestationGate.instance.bin: $<TARGET_PROPERTY:CAmkESComponent_AttestationGate_Impl_SW_AttestationGate_Attes
-- Found extra flags for SW_Filter_Filter.instance.bin: $<TARGET_PROPERTY:CAmkESComponent_Filter_Impl_SW_Filter_Filter,COMPONENT_C_FLAGS>;$<TARGET_PROPERT
-- Found extra flags for SW_Monitor_Monitor.instance.bin: $<TARGET_PROPERTY:CAmkESComponent_Monitor_Impl_SW_Monitor_Monitor,COMPONENT_C_FLAGS>;$<TARGET_PR
-- Found extra flags for SW_AttestationTester_AttestationTester.instance.bin: $<TARGET_PROPERTY:CAmkESComponent_AttestationTester_Impl_SW_AttestationTest
-- Found extra flag for SW_AttestationTester_AttestationTester.instance.bin: -L
-- Found extra flag for SW_AttestationTester_AttestationTester.instance.bin: /home/vagrant/CASE/camkes/projects/camkes/apps/CAmkES/components/Attestation
-- Found extra flag for SW_AttestationTester_AttestationTester.instance.bin:
-- CPIO test cpio_reproducible_flag PASSED
-- Configuring done
-- Generating done
-- Build files have been written to: /home/vagrant/CASE/camkes/build_CAmkES
[508/905] Building C object CMakeFiles/SW_AttestationGate_AttestationGate.instance.bin.dir/ap...CAmkES/components/AttestationGate_Impl_SW_AttestationGate
```

Completing the build:

```
-- CPIO test cpio_reproducible_flag PASSED
-- Configuring done
-- Generating done
-- Build files have been written to: /home/vagrant/CASE/camkes/build_CAmkES
[898/905] Generating capdl_spec.c
```

Simulation running in QEMU is shown in Figure 33:

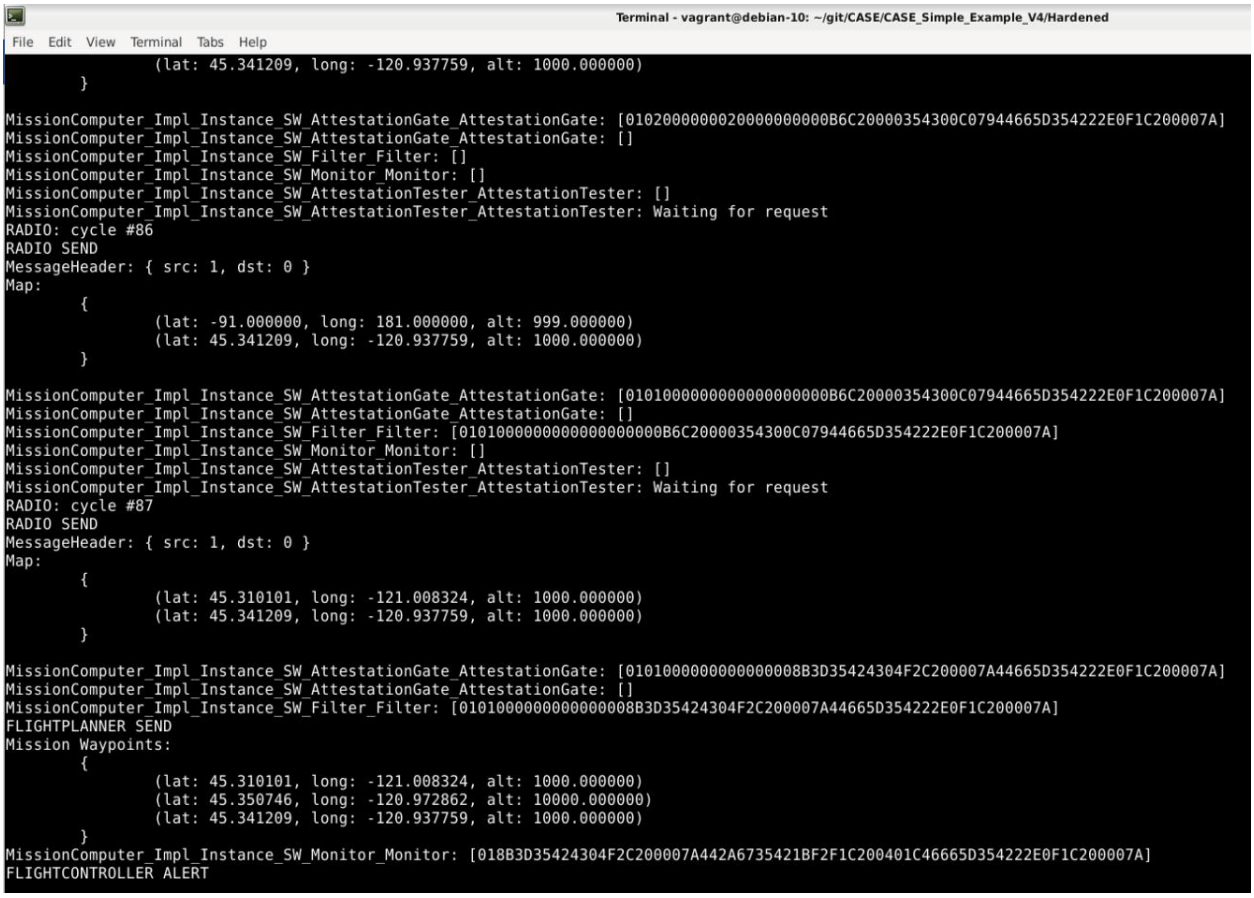


Figure 33. QEMU simulation of the hardened system

To end the QEMU session: Ctrl-a x

This is what you should see in the simulation. The Radio sends out three different messages:

- The first originates from an untrusted source and is meant to be blocked by the Attestation Gate.
- The second originates from a trusted source, but contains malformed waypoints, and is meant to be blocked by the filter.
- The third message should pass both the gate and the filter and reach the Flight Planner.

The Flight Planner has been implemented to insert a wayward waypoint into every other set of waypoints it outputs, starting from the first. If you have chosen the “pass” implementation of the AttestationTester component and the attestation mechanism has done its job, it will send a trusted ID list to the Attestation Gate (we are expecting that ID to be “1”).

We expect the Radio’s first and fourth messages to be blocked by the gate; the second and fifth messages to be blocked by the filter; the third message to be blocked by the monitor; and the sixth message to be received by the Flight Controller (along with the legitimate waypoint inserted by the Flight Planner). This pattern should repeat forever.

If you instead choose the “fail” implementation of the AttestationTester, the trusted ID list will remain empty and all messages will be blocked by the attestation gate.

8.4 Building Hardened System with Virtualized Flight Planner

Now we will build the hardened system including virtualization of the FlightPlanner. The first step is to update the Component_Type property that we discussed in Section 5.5:

```
HAMR::Component_Type => VIRTUAL_MACHINE;
```

Adding this property to the Flight Planner instructs HAMR to place the Flight Planner in a VM. At this time, HAMR’s support for VMs executing within the QEMU framework extends only to the ARM platform. This necessitates that we build our system components accordingly as we now discuss.

8.4.1 Discard previous HAMR folder

The *integrate.sh* script provided with the Hardened-VM model, assumes that the user is starting from scratch, i.e. there was no HAMR folder in place before running HAMR on the Hardened-VM model. If there was already a HAMR folder in place, we suggest saving your source files in an alternate location and discarding the HAMR folder before running HAMR on the Hardened-VM model. This will prevent the *integrate.*sh (which contains a patch) from clobbering the existing files.

8.4.2 Build the attestation mechanism for ARM

Delete the existing build folder for attestation, and rebuild the attestation mechanism as follows:

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation$ rm -rf build/ && mkdir build/ && cd build/
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation/build$ cmake ..
-- The C compiler identification is GNU 8.4.0
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/vagrant/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation/build
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation/build$ ccmake ..
```

Set up *cmake* to use the ARM compiler for building the Attestation mechanism by running *ccmake* (note the target set to *arm8* for *CAKE_FLAGS*, and the target architecture set to *aarch64*):

Steps in editing: select field and press *enter* to edit; press *c* to configure; and press *g* to generate the updated *cmake* configuration. Then, build *heli_am* as follows:

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation/build$ make heli_am
[  7%] Generating ../../heli_am.S
[ 15%] Built target heli_am.cake
Scanning dependencies of target hacl-star
[ 23%] Creating directories for 'hacl-star'
[ 23%] Performing download step (git clone) for 'hacl-star'
```

And repeat the same for *user_am*:

```
vagrant@debian-10:~/git/CASE/CASE_Simple_Example_V4/Hardened-VM/Component_Source/Attestation/build$ make user_am
[  7%] Performing update step for 'hacl-star'
[ 15%] No configure step for 'hacl-star'
[ 15%] No build step for 'hacl-star'
[ 23%] No install step for 'hacl-star'
[ 30%] Completed 'hacl-star'
[ 46%] Built target hacl-star
```

Finally, copy the *user_am* to the appropriate folder (the same filename as given in SW.aadl).

```
build$ cp user_am.S ../../Hardened/SW_AttestationTester/Pass/user_am.S
build$ cp apps/case-tool-assessment/libuser_am_c.a ../../Hardened/SW_AttestationTester/
build$
```

8.4.3 Build the gate, filter and monitor for ARM

The penultimate line of the makefile (in *Component_Sources/SW_<component_name>/make.sh*) has been modified to pass the target platform to the Cake compiler. Also note that the script assumes that the CakeML compiler is at */usr/local/bin/cake-x64-64/cake*.

```
#!/usr/bin/env bash
#
# This file is autogenerated.  Do not edit
#
set -e

export BASE_NAME=$1
CAKE=/usr/local/bin/cake-x64-64/cake
# ODROID-32 is arm7
# Xilinx-64 is arm8
# x86-64 is the default (no target)
# cake32 -w-target=arm7 --heap_size=4 --stack_size=4 < ${SCRIPT_HOME}/src/${BASE_NAME}.cml > ${SCRIPT_HOME}/src/${BASE_NAME}.S
# ${CAKE} --heap_size=4 --stack_size=4 < ${BASE_NAME}.cml > ${BASE_NAME}.S
${CAKE} --target=arm8 --heap_size=10 --stack_size=10 < ${BASE_NAME}.cml > ${BASE_NAME}.S
sed -i 's/cdecl(main)/cdecl(run)/' ${BASE_NAME}.S
~
~
```

This make script is available in the folders for the Filter, Monitor and the Attestation Gate. Section 8.3 describes the build process for these components.

8.4.4 VM Application for the Flight Planner

The source code for the Flight Planner application to run in the VM is at *Component_Source/Hardened/SW_FlightPlanner/vmFlightPlanner/*. This will be copied over by the *integrate.sh* script that is run later in the build process.

8.4.5 Rest of the build steps

The rest of the build follows the steps in building the Hardened model (without virtualization for the Flight Planner) described in Section 8.3:

1. Run the HAMR transpiler:

   ```
   ./HAMR_Simple_V4/bin/transpile-sel4.sh
   ```

2. Run the integrate script to copy over source code to the HAMR folder, and setup HAMR's build environment:

   ```
   ./integrate.sh
   ```

3. Run the QEMU simulation:

```
./run.sh
```

The QEMU simulation (as before) starts each of the components, including the Flight Planner VM. But, the Flight Planner VM takes much longer to boot than the rest of the components. We've estimated it to take approximately 120 seconds based on the execution schedule provide (*domain-schedule/schedule.c*; and copied over to *HAMR_Simple_V4/CAmkES/kernel/domain_schedule.c*), so you should see VM bootup messages interspersed with the messages from the rest of the system components. Once booted, the Flight Planner VM will display a login prompt. Enter username as *root*, and start the Flight Planner application by typing *vmFlightPlanner*.

Your display will look like Figure 34.

```
[   18.864959] Event Bar (dev-0) initalised
[   18.881619] 2 Dataports MissionComputer_Impl_Instance_SW_Monitor_Monitor: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: Waiting for request
RADIO: cycle #37
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: []
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_Filter_Filter: []
(dev-0) initalised
[   22.004885] Event Bar (dev-1) initalised
[   22.028682] 2 Dataports (dev-1) initalised
[   22.060924] Event Bar (dev-2) initalised
[   22.093304] 2 Dataports (dev-2) initalised

Welcome to Buildroot
buildroot login: MissionComputer_Impl_Instance_SW_Monitor_Monitor: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: [01BFE5889243049065FF95A245677528]
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: Recieved request: 0xBFE5889243049065FF95A24567752834
RADIO: cycle #38
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: [01010000007B0A202022636F6E7374727563746F7222203A202472
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: Received id: 01000000
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: Appraisal succeeded
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: Adding 0x01000000 to the whitelist
MissionComputer_Impl_Instance_SW_AttestationManager_AttestationManager: 0x01000000 already in the whitelist
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: [01000000000000000000000000010000]
MissionComputer_Impl_Instance_SW_Filter_Filter: []
root
# MissionComputer_Impl_Instance_SW_Monitor_Monitor: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: Waiting for request
RADIO: cycle #39
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_Filter_Filter: []
# vmFlightPlanner
Successfully setup /dev/uio0
Successfully setup /dev/uio1
Successfully setup /dev/uio2
FLIGHTPLANNER
MissionComputer_Impl_Instance_SW_Monitor_Monitor: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: []
MissionComputer_Impl_Instance_SW_AttestationTester_AttestationTester: Waiting for request
RADIO: cycle #40
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_AttestationGate_AttestationGate: []
MissionComputer_Impl_Instance_SW_Filter_Filter: []
FLIGHTPLANNER
MissionComputer_Impl_Instance_SW_Monitor_Monitor: []
```

Figure 34. QEMU simulation with FlightPlannerVM

Once the Flight Planner application has started, the rest of the simulation should proceed as described in Section 8.3. As before, *Ctrl-a x* ends the QEMU simulation.