Integrated AADL Analysis

Tutorials (Ver. 0.1.0 Rev. 10-Sep-2018)

This material is based upon work supported by the U.S. Army Research Development and Engineering Command (RDECOM), Aviation Missile Research Development and Engineering Center (AMRDEC), Aviation Development Directorate (ADD) under contract no. W911W6-17-D-0003. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Army RDECOM or AMRDEC.

Table of Contents

Overview

This guide provides a set of tutorials touching on a variety of the Architecture Analysis and Design Language (AADL) analyses. This guide is part of a training package that includes features of AADL and the Future Airborne Capability Environment (FACE) Technical Standard. The examples in this training package are based on the Basic Avionics Lightweight Source Archetype (BALSA) example provided by the FACE™ Consortium and reference an AADL model of BALSA included in the training package. This guide uses the Open Source AADL Tool Environment (OSATE) as its development environment for AADL modeling.

For updates to this document and related information, see Tools, Training, and Reference Materials for the FACE Technical Standard [https://www.adventiumlabs.com/camet/face]

Additional FACE and AADL Resources

- Additional FACE resources can be found at: FACE Documents [http://www.opengroup.org/face/information]
- Additional information about BALSA can be found at: BALSA Overview [https://publications.opengroup.org/d207]
- Additional information about OSATE can be found at: About OSATE [http://osate.org/about-osate.html]
- Additional information about Real-Time Executive for Multiprocessor Systems (RTEMS) can be found at: RTEMS Documentation [https://docs.rtems.org/branches/master/]
- Additional resources for the combined use of AADL and the FACE(TM) Technical Standard can be found at: Tools, Training, and Reference Materials for the FACE Technical Standard [https://adventiumlabs.com/CAMET/FACE]
- Additional AADL resources for CAMET subscribers can be found at: AADL Resources [https://camet.adventium.com/CAMET/CAMET/wikis/support/aadl-resources]
- Framework for Analysis of Schedulability, Timing and Resources (FASTAR) tool suite plugin and documentation (for CAMET subscribers): FASTAR [https://camet.adventium.com/CAMET/CAMET/wikis/tool_pages/FASTAR]
- Continuous Virtual Integration Toolkit (CVIT) tool documentation (for CAMET subscribers): CVIT [https://camet.adventium.com/CAMET/CAMET/wikis/tool_pages/continuous-virtual-integration]

Setup

- OSATE is available for download here: Latest stable OSATE version [https://osate-build.sei.cmu.edu/download/osate/stable/latest/products/] (this guide was tested with version 2.3.4)
- OSATE installation instructions can be found here: OSATE Installation [http://osate.org/download-and-install.html#new-installation]

Use these steps to add the prerequisite models to your OSATE workspace

1. Use the installation instructions from the OSATE site to download and install OSATE.
2. Download the prerequisite model archive `camet-training.zip`.

In addition to the models required to perform the training, this archive contains a "solution" project for each lesson for your reference. The solution projects are located in `training_materials` directory and are named for their relevent lesson number.

Use these steps to install the Basswood virtual machine on your workstation

1. Basswood runs on a virtual machine. Download and install VirtualBox (v5.2 or higher) from here: VirtualBox [https://www.virtualbox.org/]

2. Decompress the Basswood VM directory.

3. In the VirtualBox Manager window, go to Machine and select Add.... Navigate to the Basswood VM image `Basswood.vbox` and click Open.

4. The Basswood VM will be listed in the VirtualBox Manager window. Select it and click Settings to open the settings for Basswood.

5. In the Settings window under General go to Advanced and make sure bidirectional is selected for Shared Clipboard and Drag'n'Drop. Click Ok. This will allow you to easily move files between your workstation and the Basswood virtual machine.

6. Click Start to boot up Basswood.

7. Within the Basswood virtual machine you will be prompted to log in. The username and passwood are both `basswood`.

Lesson 1. The BALSA Model

Prerequisites

- Download and unzip the latest stable release of OSATE on your workstation (see the section called “Setup”)
- Download the prerequisite models on your workstation (see the section called “Setup”)
- Have a basic understanding of AADL and BALSA (see the section called “Additional FACE and AADL Resources”)

Summary

In this section, you will learn how to:

1. Open and navigate in OSATE
2. Install the FACE Data Model to AADL Translator
3. Use the FACE Data Model to AADL Translator
4. Explore the BALSA AADL model using OSATE
5. Use OSATE tools to create and edit a diagram of your BALSA model

Navigating in OSATE

Open OSATE by clicking on Osate.exe in the unzipped OSATE archive downloaded as a prerequisite for this training (see the section called “Setup”). OSATE is an open-source Eclipse-based AADL editor and will be used for the AADL work in this training. OSATE has a number of possible views that are useful for navigating your AADL workspace. The left side of the window is the **AADL Navigator** view, which lists all of the projects in your workspace. The far right side of the screen is the **Outline** view, which displays all of the elements in the model you have open currently. If these toolbars are missing go to Window>Show View and select Outline and Project Explorer.

1. Create a new project by going to File>New>AADL Project

2. Name it *BALSA* and click Finish

3. Locate your `balsa.face` file in the model archive (`/models/balsa.face`) and add it to your new BALSA project by dragging it into the `BALSA` project in the **AADL Navigator** view.

You can open the `balsa.face` file by double clicking it in the **AADL Navigator**. The center portion of your OSATE window is where the contents of your models are displayed. Since the `balsa.face` file is not written in AADL it is displayed as a "read-only" system hierarchy.

What is BALSA?

Basic Avionics Lightweight Source Archetype (BALSA) is a working example of a system of FACE-aligned components. It is a simple avionics control system with four units of portability (UoPs). BALSA has three platform specific service segments (PSSS): an Embedded GPS/INS (EGI) controller that outputs position and altitude, an Aircraft Config service that outputs callsign and aircraft ID, and an Automatic Dependent Surveillance-Broadcast (ADS-B) communication output component. There is a portable components segment (PCS) called the ATC manager that combines the outputs of the EGI and Aircraft Config into information for the ADS-B component. The transport service segment (TSS) routes messages between the UoPs. You will need to install the FACE Data Model to AADL Translator OSATE plugin to translate the `balsa.face` file into its AADL equivalent.

Installing the FACE Data Model to AADL Translator

You can install supplemental components, such as the FACE Data Model to AADL Translator, from within OSATE itself by following these steps.

1. Select Help>Install Additional OSATE Components.

2. Select FACE Data Model to AADL Translator in the popup window.

3. Click Finish to initiate the installation.

4. An Install dialog will appear with the Translator already selected. Click Next to continue.

5. After reviewing the installation details, click Next again.

6. Accept the terms of the license agreement, then click Finish to complete the installation.

7. A security dialog will warn that you are attempting to install software with unsigned content. Click on Install anyway to continue.

8. A new dialog will instruct you to restart OSATE for your software changes to take effect. Unless you have other tasks to complete first, click on Restart now.

9. After OSATE restarts, you can verify a correct installation by selecting Help>About OSATE2, click on Installation Details, and confirm that FACE Data Model to AADL Translator appears in the list of Installed Software.

Using the FACE Data Model to AADL Translator

Now that the translator is installed, you can use it to translate the `balsa.face` file into AADL. To use the translator, navigate to the `balsa.face` file in the **AADL Navigator** toolbar and right click. Select the option translate to AADL from the menu. The translator will produce a folder in your BALSA project named `model-gen` that contains four AADL files (see Figure 1). You will learn more about the contents of these files in the next section. If your generated model is filled with warnings or errors try re-building your model by going to Project>Clean... This will take a few seconds will resolve any errors in your model.

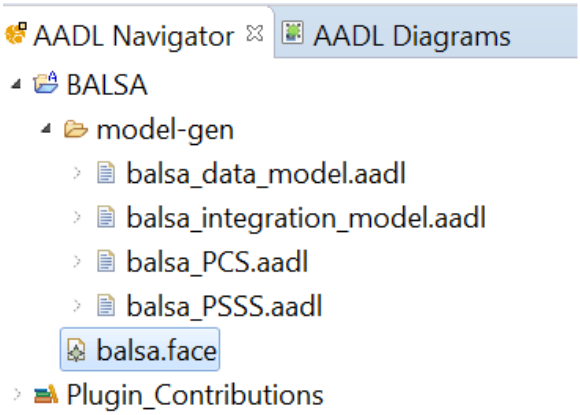


Figure 1. Translated BALSA Model Shown in the OSATE AADL Navigator

Exploring the BALSA AADL model

In the section called “Using the FACE Data Model to AADL Translator” you generated a BALSA model in AADL using the FACE Data Model to AADL Translator plugin. The BALSA UoPs are modeled as thread groups each in their own process. You are going to add properties and data flows to this model in the section called “Lesson 2. Modifying the BALSA AADL Model” to perform latency analysis.

The autogenerated BALSA AADL model is organized into four files: `balsa_data_model.aadl`, `balsa_integration_model.aadl`, `balsa_PCS.aadl`, and `balsa_PSSS.aadl`.

`balsa_data_model.aadl` contains the BALSA data type declarations and implementations. Each data type declaration has FACE-specific properties that correspond to its FACE realization tier and its UUID. `balsa_PCS.aadl` and `balsa_PSSS.aadl` contain the `thread group`, `thread`, and `process` declarations and implementations for the four BALSA UoPs; `ATCManager`, `ADSB`, `AirConfg`, and `EGI`. Properties on the UoPs such as output rates and UUIDs that are defined in the `balsa.face` file are automatically populated in the autogenerated AADL model. Each `thread group` contains a single empty thread with a period of 1 second by default (see Example 1).

```
36  thread group implementation ATCManager.impl
37   subcomponents
38    thread0: thread {
39     Period => 1 sec;
40    };
41  end ATCManager.impl;

From [balsa_PCS.aadl]
```

Example 1. A Thread Declaration Generated by the FACE Data Model to AADL Translator

`balsa_integration_model.aadl` contains the top-level integrated system with the relevant attributes from the `balsa.face` model. The `balsa_integration_model.aadl` system implementation named `BALSA_Integration_Model.impl` will be used for all of the analysis performed during this training. This system implementation has eight subcomponents; four UoPs, a bus named `UDP` as well as three abstract `transporter` components. These `transporter` components are the equivalent of the FACE notion of a TSS. The translator generates a `transporter` component for each interface between UoPs. These `transporters` can be packaged into one or more abstract TSS components by the user depending on the configuration of their specific system. BALSA traditionally has one TSS component. However, the packaging of the `transporters` is not specifically relevant to

the types of analysis covered in this tutorial, so it will not be included in the training content. There will be an optional tutorial at the end of the section called “Lesson 2. Modifying the BALSA AADL Model” with an example of how to package the TSS, but examples in later lessons will not include it. Next you will use the OSATE graphical editor to generate a diagram of your AADL BALSA model similar to Figure 2.

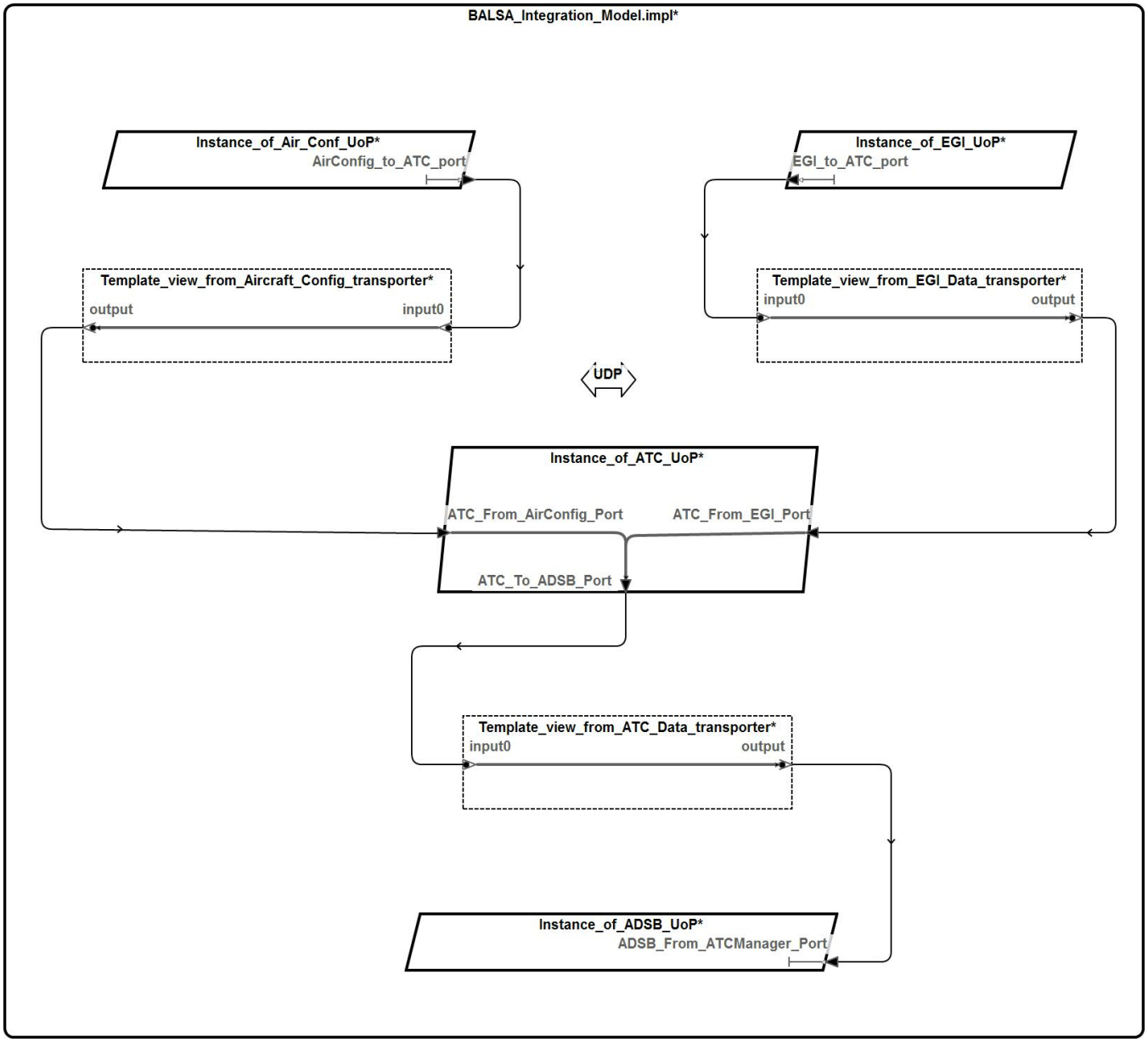


Figure 2. AADL Graphical Model of the BALSA Architecture

Using OSATE tools to Create and View Diagrams

OSATE has the ability to automatically generate diagrams of component implementations from AADL code. AADL diagrams serve as a useful visual representation of the system defined in the code.

To generate a diagram of the BALSA system:

1. Open the `balsa_integration_model.aadl` file by double clicking it in the **AADL Navigator**.

2. Make sure that you have the **Outline** view open (see the section called “Navigating in OSATE”) and navigate to the `BALSA_Integration_Model.impl` system implementation.

3. Right click the system implementation and choose Create Diagram...

4. In the opened window, name the diagram `BALSA_Autogen`, select Structure Diagram and click OK.

Your diagram should now be populated with your integrated BALSA system (see Figure 3).

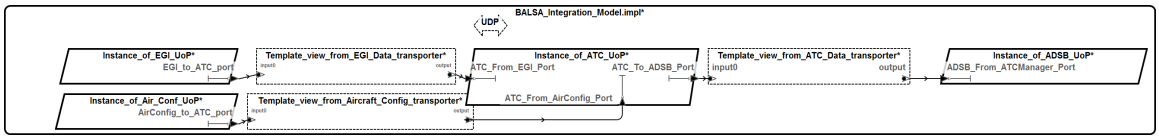


Figure 3. AADL BALSA Architecture Diagram after Editing

AADL Diagrams are generated in a directory named `diagrams`. Diagrams can be edited, saved, and exported as images. Change the sizes and locations of the diagram elements as necessary by clicking and dragging model elements (see Figure 3). For more information about AADL diagrams go to Help>Help Contents>OSATE Graphical Editor Documentation.

Lesson 2. Modifying the BALSA AADL Model

Prerequisites

- Complete the section called “Lesson 1. The BALSA Model” and have the translated AADL BALSA model in your OSATE workspace
- Download and import the prerequisite model archive in your OSATE workspace (see the section called “Setup”)

Summary

In this section, you will learn how to:

1. Add data flow specifications to the BALSA AADL model
2. Add properties to an AADL model
3. Add threads to the AADL BALSA model
4. Add flow latency properties to an AADL model
5. Perform latency analysis on an AADL model
6. Optional: Package the transporters into a TSS

Introduction to Data Flows in AADL

AADL requires the modeler to explicitly declare the path(s) through the system through which data travels. `Flows` are the AADL representation of how information flows through a system. In AADL, there are four types of data flows: `flow sinks`, `flow sources`, `flow paths`, and `end to end flows`. A `flow source` denotes the origin of a data flow and a `flow sink` denotes a data flow termination. A `flow path` defines the path through a component that a data flow follows from input to output. `End to end flows` define the paths through the system that information travels from a source component to a sink component. Like components, flows have both a declaration as well as an implementation.

The AADL model you generated in the section called “Lesson 1. The BALSA Model” using the FACE Data Model to AADL Translator takes every UoP component port and generates a `flow sink` or `flow source` for it depending on the port direction. You will have to add `flow paths` through the `transporters` as well as the `ATC` UoP to build the `end to end flows` necessary for timing analysis. In the case of BALSA, you will define the paths through the system that begin in the `EGI` and `AirConfig` UoPs and terminate in the `ADSB` UoP.

Adding Threads to the AADL BALSA Model

Navigate to the `ATCManager` thread group declaration in `balsa_PCS.aadl`. The thread declaration generated by the translator is empty (see Example 2).

```
36  thread group implementation ATCManager.impl
37   subcomponents
38    thread0: thread {
39     Period => 1 sec;
40    };
41  end ATCManager.impl;

From [balsa_PCS.aadl]
```

Example 2. Empty Thread Declaration Generated by the FACE Data Model to AADL Translator

Now compare it to the thread `executable_thread` in `BALSA_Software.aadl` in the `BALSA_Software` prerequisite project. This thread has input and output data ports as well as a `Compute_Execution_Time` property declaration (see Example 3).

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/BALSA_Software/BALSA_Software.aadl]
```

Example 3. A Thread with Worst Case Execution Time Specified

Move the `BALSA_Software.aadl` model file (located in the prerequisite model archive under `/training_materials/BALSA_Software`) into your BALSA project by dragging it into the `model-gen` folder. Go back to `ATCManager` in `balsa_PCS.aadl` and add `BALSA_Software::executable_thread.impl` for the thread name in the `subcomponents` declaration (see Example 4). Add `with BALSA_Software;` at the top of `balsa_PCS.aadl` to get rid of the model error that appears.

```
50  thread group implementation ATCManager.impl
51   subcomponents
52    thread0: thread BALSA_Software::executable_thread.impl{
53     Period => 1 sec;
54    };

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 4. ATCManager Thread Group with a Thread Implementation

Now that the thread declaration is no longer empty, add `connections` after the `subcomponents` of the ATCManager thread group (see Example 5).

```
57   connections
58    c01: feature ATC_From_AirConfig_Port -> thread0.input;
59    c02: feature ATC_From_EGI_Port -> thread0.input;
60    c03: feature thread0.output -> ATC_To_ADSB_Port;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 5. Thread Connections in the ATCManager Thread Group

Add a `flows` section above the `properties` section in the thread group declaration and create flow declarations between the inputs and the output (see Example 6).

```
37   flows
38    AirConfig_To_ADSB_path: flow path ATC_From_AirConfig_Port ->
39     ATC_To_ADSB_Port;
40    EGI_To_ADSB_path: flow path ATC_From_EGI_Port -> ATC_To_ADSB_Port;
41
42   properties

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 6. Flow Declarations for ATCManager

Create implementations of these flow paths in the `ATCManager.impl` thread group implementation that go through the thread (see Example 7). Note that the flow path implementations should have the same name as the declarations you just created.

```
63   flows
64    AirConfig_To_ADSB_path: flow path ATC_From_AirConfig_Port -> c01 -> thread0
65     -> c03 -> ATC_To_ADSB_Port;
66    EGI_To_ADSB_path: flow path ATC_From_EGI_Port -> c02 -> thread0 -> c03 ->
67     ATC_To_ADSB_Port;
68

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 7. Flow Implementations for ATCManager

Replace the `flow sink` and `flow source` declarations in the process `ATCManager_process` with `flow paths` between the two input ports and the output port (see Example 8).

```
90   flows
91    AirConfig_To_ADSB_path: flow path ATC_From_AirConfig_Port ->
92     ATC_To_ADSB_Port;
93    EGI_To_ADSB_path: flow path ATC_From_EGI_Port -> ATC_To_ADSB_Port;
94
95  end ATCManager_process;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 8. Flow Path Declarations for ATCManager

Now navigate to the implementation of the `ATCManager` process and create implementations of the flow paths that travel through the `ATCManager` thread group (see Example 9). Note that the flow path implementations should have the same name as the declarations you just created. You are adding detail to the original declaration now that more detail about the system composition is available (i.e. subcomponents). The declaration defines the start and end point of a flow, the implementation declares how it travels through the system.

```
105   flows
106    AirConfig_To_ADSB_path: flow path ATC_From_Airconfig_Port ->
107     ATC_From_AirConfig_Port_connection -> ATCManager.AirConfig_To_ADSB_path
108     -> ATC_To_ADSB_Port_connection -> ATC_To_ADSB_Port;
109
110    EGI_To_ADSB_path: flow path ATC_From_EGI_Port -> ATC_From_EGI_Port_connection
111      -> ATCManager.EGI_To_ADSB_path -> ATC_To_ADSB_Port_connection ->
112      ATC_To_ADSB_Port;
113
114  end ATCManager_process.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PCS.aadl]
```

Example 9. Flow Path Implementations for ATCManager

Repeat this process for `balsa_PSSS.aadl` in the `ADSB` thread group declaration above the `properties` (see Example 10),

```
27   flows
28    flow_sink: flow sink ADSB_From_ATCManager_Port;
29
30   properties

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 10. ADSB Flow Declaration

`ADSB.impl` thread group implementation (see Example 11),

```
41  thread group implementation ADSB.impl
42   subcomponents
43    thread0: thread BALSA_software::executable_thread.impl {
44     Period => 1 sec;
45    };
46   connections
47    c01: feature ADSB_From_ATCManager_Port -> thread0.input;
48   flows
49    flow_sink: flow sink ADSB_From_ATCManager_Port -> c01 -> thread0;
50  end ADSB.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 11. Update the thread and declare thread flow implementations for ADSB

`ADSB_process.impl` process implementation (see Example 12),

```
63  process implementation ADSB_process.impl
64   subcomponents
65    ADSB: thread group ADSB.impl;
66   connections
67    ADSB_From_ATCManager_Port_connection: port ADSB_From_ATCManager_Port ->
68     ADSB.ADSB_From_ATCManager_Port;
69  flows
70    ADSB_From_ATCManager_Port_sink: flow sink ADSB_From_ATCManager_Port ->
71     ADSB_From_ATCManager_Port_connection -> ADSB.flow_sink;
72  end ADSB_process.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 12. Declare flows for the ADSB process

`AirConfig` thread group declaration (see Example 13),

```
83   flows
84    AirConfig_to_ATC_port_source: flow source AirConfig_to_ATC_port;
85
86   properties

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 13. Declare a flow for AirConfig

`AirConfig.impl` thread group implementation (see Example 14),

```
93  thread group implementation AirConfig.impl
94   subcomponents
95    thread0: thread BALSA_Software::executable_thread{
96     Period => 1 sec;
97    };
98   connections
99    c01: feature thread0.output -> AirConfig_to_ATC_port;
100
101   flows
102    AirConfig_to_ATC_port_source: flow source thread0 -> c01 ->
103     AirConfig_to_ATC_port;
104  end AirConfig.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 14. Thread Flow Implementations for AirConfig

A `process` provides memory space for thread groups and threads. The `AirConfig_process.impl` process implementation (see Example 15) provides memory space for the AirConfig thread group. Flows originating or terminating in a subcomponent of a process must have an explicit flow segment defining how they enter and exit the process.

```
117  process implementation AirConfig_process.impl
```

```
118   subcomponents
119    AirConfig: thread group AirConfig.impl;
120   connections
121    AirConfig_to_ATC_port_connection: port AirConfig.AirConfig_to_ATC_port ->
122     AirConfig_to_ATC_port;
123   flows
124    AirConfig_to_ATC_port_source: flow source AirConfig.AirConfig_to_ATC_port_source
125     -> AirConfig_to_ATC_port_connection -> AirConfig_to_ATC_port;
126  end AirConfig_process.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 15. Flows for the AirConfig Process

`EGI` thread group declaration above the `properties` (see Example 16),

```
137    flows
138   EGI_to_ATC_port_source: flow source EGI_to_ATC_port;
139
140   properties

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 16. Declare a flow for EGI

`EGI.impl` thread group implementation (see Example 17),

```
147  thread group implementation EGI.impl
148   subcomponents
149    thread0: thread BALSA_Software::executable_thread{
150     Period => 1 sec;
151    };
152   connections
153    c01: feature thread0.output -> EGI_to_ATC_port;
154   flows
155    EGI_to_ATC_port_source: flow source thread0 -> c01 -> EGI_to_ATC_port;
156  end EGI.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 17. Thread Flow Implementations for EGI

`EGI` process implementation (see Example 18),

```
169  process implementation EGI_process.impl
170   subcomponents
171    EGI: thread group EGI.impl;
172   connections
173    EGI_to_ATC_port_connection: port EGI.EGI_to_ATC_port -> EGI_to_ATC_port;
174   flows
175    EGI_to_ATC_port_source: flow source EGI.EGI_to_ATC_port_source ->
```

```
176     EGI_to_ATC_port_connection -> EGI_to_ATC_port;
177  end EGI_process.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 18. Flows for the EGI process

These components are only sources or sinks of data flows, so each thread group and process only needs one connection and one data flow each. A `flow source` component is the origin of data (e.g. sensors) and a `flow sink` is the terminal component in the information flow (e.g. actuators, memory, etc.). Information in AADL systems flows from a `flow source` to a `flow sink` component. `Flow paths` denote intermediary components that pass the information on to another component.

Adding End to End Flows to the AADL BALSA Model

Next you will define flow paths through the `transporter` components. These are located in `balsa_integration_model.aadl`. Unlike the UoP components, there are no flow declarations for the `transporters` generated by the translator. Add a `flows` section between the `features` and `properties` sections of each of the three abstract transporter declarations with a `flow path` between `input0` and `output` (see Example 19).

```
98      flows
99    flow_path: flow path input0 -> output;
100
101   properties

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_integration_model.aadl]
```

Example 19. Flow Path for each Transporter

With all of the component flows declared, you are ready to add the `end to end flows` through the integrated system necessary for latency analysis. Unlike the other types of flow declarations, `end to end flows` only have an implementation. Navigate to `BALSA_Integration_Model.impl` in `balsa_integration_model.aadl` and add a `flows` section below the connections. The `end to end flow` is declared from source to sink following the logical flow from one component flow implementation to the next through connections. Examples of the `end to end flow` declarations are shown in Example 20. Keep in mind that yours will differ slightly depending on what you named the flow paths you created in the `ATCManager` and `transporters`. Create two `end to end flows`; one from `EGI` through `ATC` to `ADSB` and one from `AirConfig` through `ATC` to `ADSB` (see Example 20). Notice that flow declarations are called using the **component_name.flow_name** syntax while connections are called by name only.

```
70     flows
71    ETE_EGI: end to end flow Instance_of_EGI_UoP.EGI_to_ATC_port_source ->
72     connection0 -> Template_view_from_EGI_Data_transporter.flow_path ->
73     connection1 -> Instance_of_ATC_UoP.EGI_To_ADSB_path -> connection2
74     -> Template_view_from_ATC_Data_transporter.flow_path -> connection3
75     -> Instance_of_ADSB_UoP.ADSB_From_ATCManager_Port_sink;
76
77    ETE_AirConfig: end to end flow
78     Instance_of_Air_Conf_UoP.AirConfig_to_ATC_port_source -> connection4
79      -> Template_view_from_Aircraft_Config_transporter.flow_path ->
80     connection5 -> Instance_of_ATC_UoP.AirConfig_To_ADSB_path -> connection2 ->
81     Template_view_from_ATC_Data_transporter.flow_path -> connection3 ->
```

```
82      Instance_of_ADSB_UoP.ADSB_From_ATCManager_Port_sink;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_integration_model.aadl]
```

Example 20. End to End Flows

You will return to these `end to end flows` later when you are performing latency analysis. Generate another diagram of your BALSA integration model and name it `BALSA_with_Flows` (see Figure 4). Notice that you will have to edit this diagram as you did in the section called “Using OSATE tools to Create and View Diagrams”.

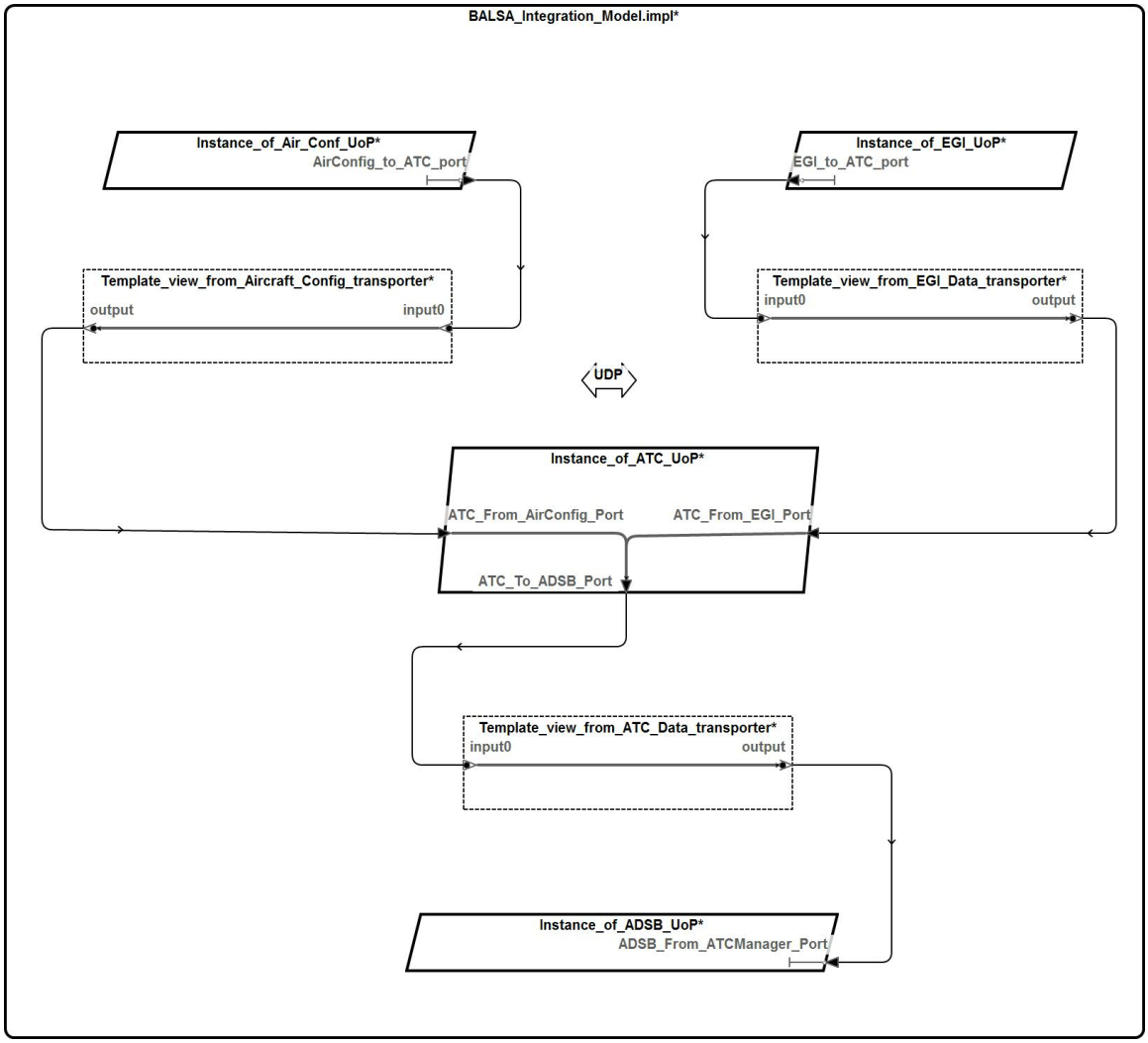


Figure 4. Edited BALSA Integration Model

If your data flows are defined correctly then you should see a clear data flow path through your BALSA system. Any sections that are missing in your diagram indicate that something is missing or incorrect in your flow declarations.

Adding Properties to an AADL Model

AADL model elements can be tagged with properties used for various types of analysis. Property types are declared in property sets, which are AADL files that define the attributes of properties such as units, types, and ranges. AADL comes with some pre-defined property sets located in the `Plugin_Contributions` directory. Users can write

their own property sets to use with their models or reference these predefined property sets. To add a property to a model, you must first include the name of the property set at the top of your model in a `with` clause in the same way that you would reference an AADL model.

To add properties to an AADL component, add a `properties` declaration within a chosen element (see Example 21). Properties are declared using the syntax of *Property_Set_Name::Property*. As mentioned in earlier sections, the translator automatically populates the model with some properties inferred from the FACE model.

```
26   properties
27    FACE::Segment => PSSS;
28    FACE::Profile => safety;
29    FACE::UUID => "_hwTh9EM1EeiBlKadCQCZ8Q";

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L1_balsa/balsa_PSSS.aadl]
```

Example 21. Sample Properties Generated from the FACE data model

Properties can be added for the purpose of model annotation, analysis, or both. Add `Code_Size` properties to the four UoP thread groups in `balsa_PCS.aadl` and `balsa_PSSS.aadl` with reasonable values for BALSA (e.g. 1000 bytes). `Code_Size` is from one of the predefined property sets, so you do not have to include the property set name for these declarations. Properties like `Code_Size` enable quanititaive analysis to be performed by OSATE plugins (see Example 22).

```
37    Code_Size => 1000 bytes;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_PSSS.aadl]
```

Example 22. Add a code size property

Performing Latency Analysis on the AADL BALSA Model

The latency analysis tool is one of the standard OSATE tools that takes timing properties in your AADL model and calculates expected latency values for full end to end system flows. The threads that you added to your AADL BALSA model in the section called “Adding Threads to the AADL BALSA Model” have an execution time property. Go to the transporters in `balsa_integration_model.aadl` and add a `Latency` property declaration on each `flow path` (see Example 23). This property is declaring that there is a latency of 1-2 ms associated with communications across the *TSS*.

```
118   properties
119    Latency => 1 ms .. 2 ms applies to flow_path;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_integration_model.aadl]
```

Example 23. Latency Property on each Flow Path

Go the the `end to end flows` in `BALSA_Integration_Model.impl` in `balsa_integration_model.aad` and add a `Latency` property declaration (see Example 24). This is your latency budget for the BALSA system, which will be compared to the latency analysis results.

```
85   properties
```

```
86    Latency => 10 ms .. 20 ms applies to ETE_EGI, ETE_Airconfig;
87  end BALSA_Integration_Model.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_integration_model.aadl]
```

Example 24. End to End Latency Budget

With these added latency properties, you have enough information to perform latency analysis on your AADL BALSA model. AADL Analysis tools use instance models as input, which resolve the attributes of the declarations and implementations into a single component representation and build the full system hierchy of the components. Generate an instance model of your BALSA system by right clicking the `BALSA_Integration_Model.impl` abstract implementation in the **Outline** view and select Instantiate System. Locate the instance model in the `instances` directory and select it. Go to Analyses>Timing>Check Flow Latency. The results of latency analysis will be added to a directory named `reports` that will be generated in the `instances` directory. Open the CSV report and verify that the *Min Actual* latency of both flows is 11.0 ms. Note that the *Max Actual* latency is 2010 ms due to the 1 second sampling delay that was generated by the translator for each thread group (see Figure 5).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Latency analysis with preference settings: asynchronous system/partition end/worst case as max compute execution time/best case as full queue min compute execution time																
2																	
3	Latency results for end-to-end flow 'ETE_EGI' of system 'BALSA_Integration_Model.impl'																
4																	
5	Result	Min Specif	Min Actua	Min Meth	Max Speci	Max Actua	Max Meth	Comments									
6	thread	Instance_of_I	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
7	connection	Instance_	0.0ms	no latency		0.0ms	no latency										
8	abstract	T	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified									
9	connection	Template	0.0ms	no latency		0.0ms	no latency										
10	thread	Instance_of_	0.0ms	sampling		1000.0ms	sampling	Assume P	Max: Worst case: Round up sampling delay to period 1000.0ms								
11	thread	Instance_of_	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
12	connection	Instance_	0.0ms	no latency		0.0ms	no latency										
13	abstract	T	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified									
14	connection	Template	0.0ms	no latency		0.0ms	no latency										
15	thread	Instance_of_	0.0ms	sampling		994.0ms	sampling	Assume P	Min: Best	Assume s	Max: Worst case: Round up sampling delay to period 1000.0ms						
16	thread	Instance_of_	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
17	Latency To	2.0ms	11.0ms		4.0ms	2010.0ms											
18	Specified End To End		10.0ms			20.0ms											
19	End to end Latency Summary																
20	WARNING	Minimum specified flow latency total 2.00ms less than expected minimum end to end latency 10.0ms (better response time)															
21	SUCCESS	Minimum actual latency total 11.0ms is greater or equal to expected minimum end to end latency 10.0ms															
22	ERROR	Maximum actual latency total 2010.0ms exceeds expected maximum end to end latency 20.0ms															
23	WARNING	Jitter of actual latency total 11.0..2010.0ms exceeds expected end to end latency jitter 10.0..20.0ms															
24																	
25																	
26																	
27	Latency results for end-to-end flow 'ETE_AirConfig' of system 'BALSA_Integration_Model.impl'																
28																	
29	Result	Min Specif	Min Actua	Min Meth	Max Speci	Max Actua	Max Meth	Comments									
30	thread	Instance_of_	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
31	connection	Instance_	0.0ms	no latency		0.0ms	no latency										
32	abstract	T	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified									
33	connection	Template	0.0ms	no latency		0.0ms	no latency										
34	thread	Instance_of_	0.0ms	sampling		1000.0ms	sampling	Assume P	Max: Worst case: Round up sampling delay to period 1000.0ms								
35	thread	Instance_of_	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
36	connection	Instance_	0.0ms	no latency		0.0ms	no latency										
37	abstract	T	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified									
38	connection	Template	0.0ms	no latency		0.0ms	no latency										
39	thread	Instance_of_	0.0ms	sampling		994.0ms	sampling	Assume P	Min: Best	Assume s	Max: Worst case: Round up sampling delay to period 1000.0ms						
40	thread	Instance_of_	3.0ms	processing time		4.0ms	processing	Using execution time as deadline was not set									
41	Latency To	2.0ms	11.0ms		4.0ms	2010.0ms											
42	Specified End To End		10.0ms			20.0ms											
43	End to end Latency Summary																
44	WARNING	Minimum specified flow latency total 2.00ms less than expected minimum end to end latency 10.0ms (better response time)															
45	SUCCESS	Minimum actual latency total 11.0ms is greater or equal to expected minimum end to end latency 10.0ms															
46	ERROR	Maximum actual latency total 2010.0ms exceeds expected maximum end to end latency 20.0ms															
47	WARNING	Jitter of actual latency total 11.0..2010.0ms exceeds expected end to end latency jitter 10.0..20.0ms															

Figure 5. Latency Report

This report shows both a failure case as well as a successful one. Line 21 shows that the actual minimum latency of 11 ms is within the specified budget, so the test was successful. However, line 22 shows an error message that states the actual maximum latency of 2010 ms is above the upper latency bound of 20 ms. This error is coupled with a warning on both the instance model as well as the latency report.

Optional: Packaging the Transporters into a single TSS Example

The transporter segment abstracts generated by the FACE Data model to AADL translator can be packaged in AADL into a single TSS. The translator is a generic tool that cannot infer how many TSS components exist in a system. It generates a transporter for every UoP interface. It is up to the user to define how these transporter segments relate to the actual TSS configuration of the system. In the case of BALSA, there is one TSS component, so you are going to package all of the transporters into one component. In your BALSA AADL project, create a copy of `balsa_integration_model.aadl` and name it `balsa_integration_TSS.aadl`. You will want this model separate from the integration model that you will be using for the remainder of the training. Rename `BALSA_Integration_Model` to `BALSA_Integration_Model_TSS` so you do not get it confused with the integration model used in the training. Create a new `TSS` abstract component and `TSS.impl` implementation and cut and paste the transporters from the subcomponents of `BALSA_Integration_Model_TSS.impl` to be subcomponents of `TSS.impl` (see Example 25). This should generate several warnings in `BALSA_Integration_Model_TSS.impl`. You will resolve these later in this section.

```
211  abstract implementation TSS.impl
212   subcomponents
213    Template_view_from_EGI_Data_transporter:
214     abstract Template_view_from_EGI_Data_transporter;
215    Template_view_from_ATC_Data_transporter:
216     abstract Template_view_from_ATC_Data_transporter;
217    Template_view_from_Aircraft_Config_transporter:
218     abstract Template_view_from_Aircraft_Config_transporter;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_TSS_package.aadl]
```

Example 25. TSS Abstract Component

Add `features` to the `TSS` abstract: one input and one output for each `transporter`. Copy the properties of each `feature` from the corresponding ports on the `transporters` to the `TSS` `features`. Create `connections` from each of the `TSS` ports to their corresponding `transporter` ports (see Example 26).

```
167  abstract TSS
168   features
169    EGI_input: in feature
170     balsa_data_model::EGI_Data_Platform.impl {
171      FACE::UUID => "_hwdLZEM1EeiBlKadCQCZ8Q";
172    };
173
174    EGI_output: out feature
175     balsa_data_model::EGI_Data_Platform.impl {
176      FACE::UUID => "_hwdLY0M1EeiBlKadCQCZ8Q";
177    };
178
179    ATC_input: in feature
180     balsa_data_model::ATC_Data_Platform.impl {
181      FACE::UUID => "_hwdLZ0M1EeiBlKadCQCZ8Q";
182    };
183
```

```
184    ATC_output: out feature
185     balsa_data_model::ATC_Data_Platform.impl {
186      FACE::UUID => "_hwdLZkM1EeiBlKadCQCZ8Q";
187    };
188
189    Aircraft_input: in feature
190     balsa_data_model::Aircraft_Config_Platform.impl {
191      FACE::UUID => "_hwdLakM1EeiBlKadCQCZ8Q";
192    };
193
194    Aircraft_output: out feature
195     balsa_data_model::Aircraft_Config_Platform.impl {
196      FACE::UUID => "_hwdLaUM1EeiBlKadCQCZ8Q";
197    };
```

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_TSS_package.aadl]
```

Example 26. TSS Features

Add `flow path` declarations and implementations that start and end at the `TSS` ports and go through your newly created connections and the relevant transporter (see Example 27 and Example 28).

```
200   flows
201    EGI_flow_path: flow path EGI_input -> EGI_output;
202
203    ATC_flow_path: flow path ATC_input -> ATC_output;
204
205    Aircraft_flow_path: flow path Aircraft_input ->
206     Aircraft_output;
```

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_TSS_package.aadl]
```

Example 27. TSS Flow Paths

```
230   flows
231    EGI_flow_path: flow path EGI_input -> EGI_in ->
232     Template_view_from_EGI_Data_transporter.flow_path -> EGI_out
233     -> EGI_output;
234
235    ATC_flow_path: flow path ATC_input -> ATC_in ->
236     Template_view_from_ATC_Data_transporter.flow_path -> ATC_out
237     -> ATC_output;
238
239    Aircraft_flow_path: flow path Aircraft_input -> Aircraft_in ->
240     Template_view_from_Aircraft_Config_transporter.flow_path ->
241     Aircraft_out -> Aircraft_output;
```

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/L2_balsa/balsa_TSS_package.aadl]
```

Example 28. TSS Flow Path Implementations

Now navigate back to `BALSA_Integration_Model_TSS.impl` and add `TSS.impl` as a subcomponent. Replace any references to the original flows though and connections to the `transporters` in

`BALSA_Integration_Model_TSS.impl` with their equivalent flows through and connections to the `TSS` subcomponent you just added (see Example 29). This should resolve the model warnings that appeared earlier.

```
27  system implementation BALSA_Integration_Model_TSS.impl
28   subcomponents
29
30    Example_Processor: processor Example_Proc.impl;
31
32    Instance_of_ATC_UoP: process
33     balsa_PCS::ATCManager_process.impl {
34      FACE::UUID => "_hwdLUUM1EeiBlKadCQCZ8Q";
35    };
36
37    Instance_of_EGI_UoP: process
38     balsa_PSSS::EGI_process.impl {
39      FACE::UUID => "_hwdLVUM1EeiBlKadCQCZ8Q";
40    };
41
42    Instance_of_Air_Conf_UoP: process
43     balsa_PSSS::AirConfig_process.impl {
44      FACE::UUID => "_hwdLV0M1EeiBlKadCQCZ8Q";
45    };
46
47    Instance_of_ADSB_UoP: process
48     balsa_PSSS::ADSB_process.impl {
49      FACE::UUID => "_hwdLWUM1EeiBlKadCQCZ8Q";
50    };
51
52    UDP: bus {
53     FACE::UUID => "_hwdLa0M1EeiBlKadCQCZ8Q";
54    };
55
56    TSS: abstract TSS.impl;
57
58   connections
59    connection0: feature Instance_of_EGI_UoP.EGI_to_ATC_port ->
60     TSS.EGI_input {FACE::UUID => "_hwdLXEM1EeiBlKadCQCZ8Q";
61    };
62
63    connection1: feature TSS.EGI_output ->
64    Instance_of_ATC_UoP.ATC_From_EGI_Port {
65    FACE::UUID => "_hwdLXUM1EeiBlKadCQCZ8Q";
66    };
67
68    connection2: feature Instance_of_ATC_UoP.ATC_To_ADSB_Port
69    -> TSS.ATC_input {FACE::UUID => "_hwdLXkM1EeiBlKadCQCZ8Q";
70    };
71
72    connection3: feature TSS.ATC_output ->
73    Instance_of_ADSB_UoP.ADSB_From_ATCManager_Port {
74    FACE::UUID => "_hwdLX0M1EeiBlKadCQCZ8Q";
75    };
76
```

```
77    connection4: feature Instance_of_Air_Conf_UoP.AirConfig_to_ATC_port
78    -> TSS.Aircraft_input {FACE::UUID => "_hwdLYEM1EeiBlKadCQCZ8Q";
79    };
80
81    connection5: feature TSS.Aircraft_output ->
82     Instance_of_ATC_UoP.ATC_From_AirConfig_Port {
83     FACE::UUID => "_hwdLYUM1EeiBlKadCQCZ8Q";
84    };
85
86   flows
87    ETE_EGI: end to end flow Instance_of_EGI_UoP.EGI_to_ATC_port_source ->
88     connection0 -> TSS.EGI_flow_path ->
89     connection1 -> Instance_of_ATC_UoP.EGI_To_ADSB_path -> connection2
90     -> TSS.ATC_flow_path -> connection3
91     -> Instance_of_ADSB_UoP.ADSB_From_ATCManager_Port_sink;
92
93    ETE_AirConfig: end to end flow
94     Instance_of_Air_Conf_UoP.AirConfig_to_ATC_port_source -> connection4
95      -> TSS.Aircraft_flow_path ->
96     connection5 -> Instance_of_ATC_UoP.AirConfig_To_ADSB_path ->
97     connection2 -> TSS.ATC_flow_path -> connection3 ->
98     Instance_of_ADSB_UoP.ADSB_From_ATCManager_Port_sink;
99
```

From [balsa_TSS_package.aadl]

Example 29. Update flow declarations

If your model is error free, then you have successfully implemented your TSS. Generate and rearrange a diagram of this system as you did in the section called “Using OSATE tools to Create and View Diagrams” and the section called “Introduction to Data Flows in AADL” (see Figure 6).

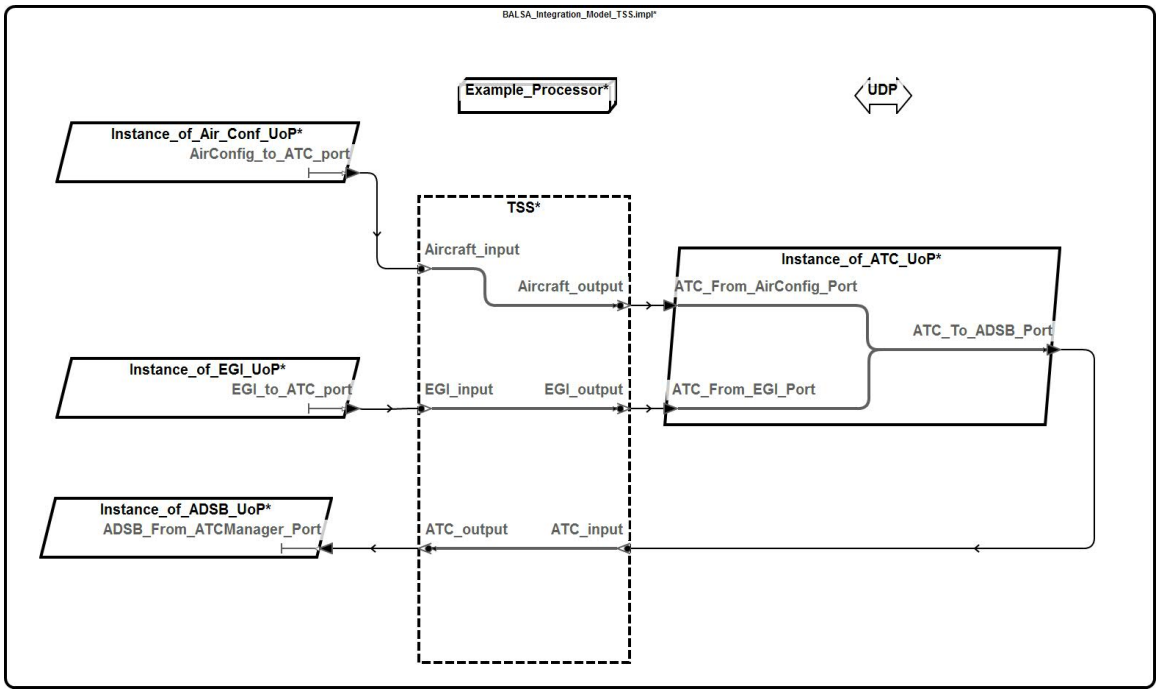


Figure 6. BALSA with TSS

Lesson 3. Executing a BALSA-Derived Demonstration System

Prerequisites

- Download and install the prerequisite model archive in your OSATE workspace (see the section called “Setup”)
- Install the Basswood Real-Time Executive for Multiprocessor System (RTEMS) build environment on your workstation (see the section called “Setup”)
- Install the FASTAR Tool Suite as well as the ARINC653 Configuration Generator Plugin (see the section called “Additional FACE and AADL Resources”)

Summary

In this lesson, you will learn how to:

1. Configure the Basswood model's real-time execution attributes
2. Generate source code from the Basswood model
3. Build and run the example application generated from the model on the real-time execution environment

Introduction to Basswood

From this lesson on, you will be working with a new AADL model named Basswood. Create an AADL project named `Basswood` and copy and paste the Basswood AADL model files from the prerequisite model archive in it (located in /data_model/training_Basswood).

Basswood is a subset of BALSA with only an ATC, AirConfig, and EGI. It is the AADL equivalent of the example you are going to work with on RTEMS (see Figure 8). The Basswood AADL model was generated using the same translator you used in the section called “Lesson 1. The BALSA Model” and then altered to more closely resemble the configuration used in the RTEMS example.

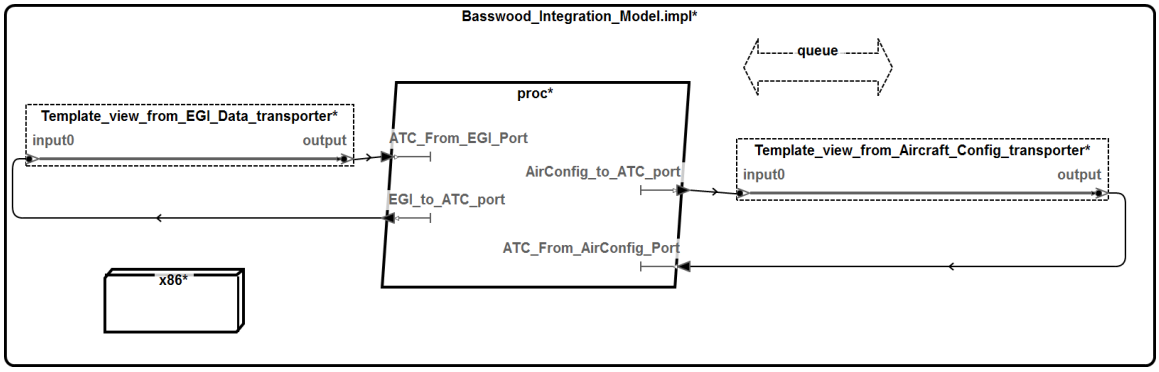


Figure 7. AADL Graphical Model of the RTEMS Basswood Demo Processor, Process, and Communication Infrastructure

The FACE Data Model to AADL Translator, by default, generates thread groups each in their own process. RTEMS does not have memory partitioning, so all of the UoP thread groups in Basswood are in the same process named `rtems.impl` in the file `basswood_schedule.aadl`.

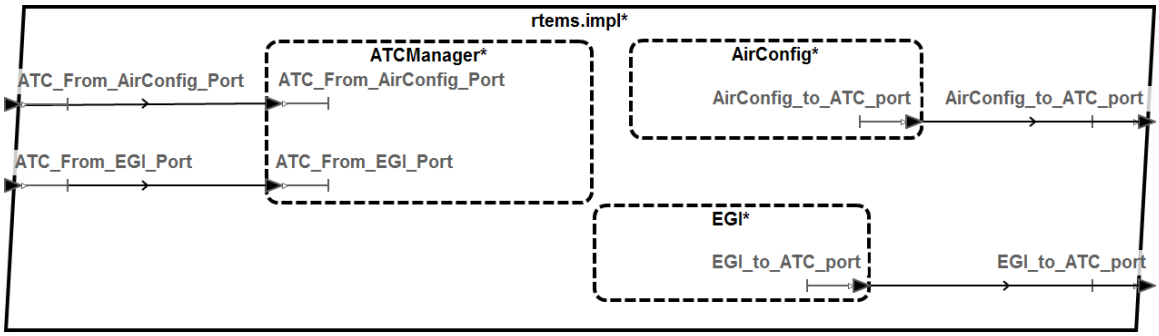


Figure 8. AADL Graphical Model of the Basswood Demo UoP Thread Groups in the rtems.impl Processor

As in the BALSA example, the top-level integration system is `Basswood_Integration_Model.impl` located in the file `basswood_integration_model`. Open this file and navigate to `Basswood_Integration_Model.impl` in the **Outline** view. Right click the implementation and select the menu option Instantiate. You will use this instance model later to generate source code for RTEMS.

Configuring Basswood Real-Time Attributes

In this example you will execute a BALSA-based training model application, named Basswood, on RTEMS. You will be using RTEMS to verify your models using a real-world system. This tutorial runs RTEMS in an emulator.

There are two task attributes that impact the real-time scheduling of the training example, applied to the producer EGI and AirConfig tasks and the consumer ATC task. The two attributes are:

1. `Timing_Properties::Period` defines the length of that time interval in which the task is periodically invoked, and

2. `Timing_Properties::Deadline` defines the maximum execution time for a single (contiguous) execution of the task.

The training example uses the rate-monotonic scheduler (RMS) employed by the RTEMS execution environment (see Figure 8 for more information). By adjusting the `period` and `deadline` attributes on the EGI, AirConfig, and ATC tasks, you can modify the timing of the task invocations. It is possible, for example, to configure the EGI, AirConfig, and ATC periods and deadlines in a way that causes messages from the EGI to get dropped, or for one of the tasks to miss an iteration deadline.

Configuring the Real-Time Execution Environment

Before you generate source code from the model, examine the project properties that define how the Basswood model is configured in the real-time execution environment. In OSATE, verify that you have the Configuration Generation plugin installed by going to Help>About OSATE2>Installation Details. Look for a plugin named **ARINC653 Configuration Generator** under the Installed Software tab. If this plugin is not listed, follow the installation instructions found in the LynxOS-178B Configuration Generation User Guide (see the section called “Setup”). Invoke the Preferences menu under Window and select RTOS Config. For the Target RTOS property, select RTEMS. The remainder of the properties on this screen address alternative RTOS configurations. Click Apply and Close.

Generating Source Code and Running the Real-Time Application

To generate RTOS source code for the model, in OSATE generate an instance of `Basswood_Integration_Model.impl` in `basswood_integration_model` as you did in the section called “Lesson 2. Modifying the BALSA AADL Model”. Select the instance file in the **AADL Navigator** and select the Generate RTOS Configuration menu option under the RTOS menu. A dialog will appear when the code generation is complete.

The generated source file named `scheduling.c` identifies the scheduling parameters of the EGI, AirConfig, and ATC components. This file is generated in the same directory as the Basswood AADL files.

Next, start the Basswood VirtualBox virtual machine (VM), which contains the RTEMS execution environment build in a Linux Ubuntu installation (See the section called “Setup”). Note that in the following examples the shorthand `~/` is occasionally used to denote the virtual machine path `/home/basswood` Double-click on the Basswood image, and when the login prompt appears, type the password `basswood` (lowercase) (See Figure 9).

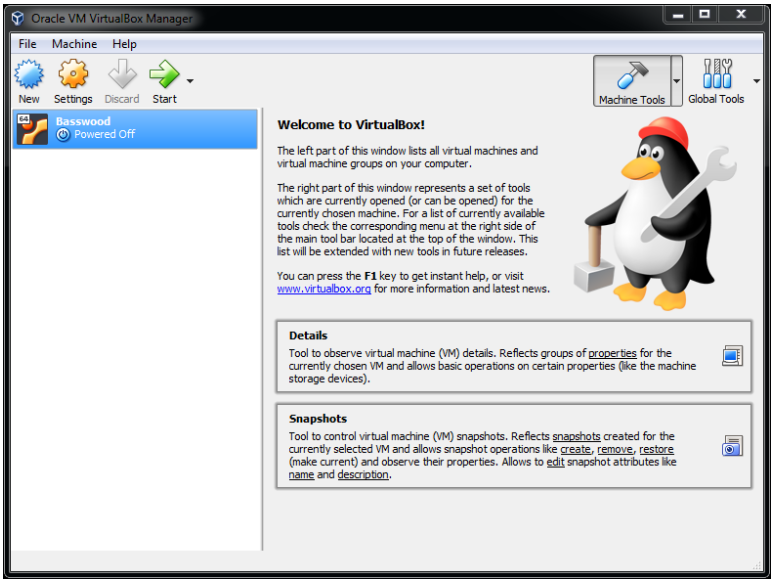


Figure 9. Basswood Virtual Machine

Share your OSATE workspace with the Basswood VM by going to Machine>Settings.... Under Shared Folders click the blue folder icon on the right side (see Figure 10).

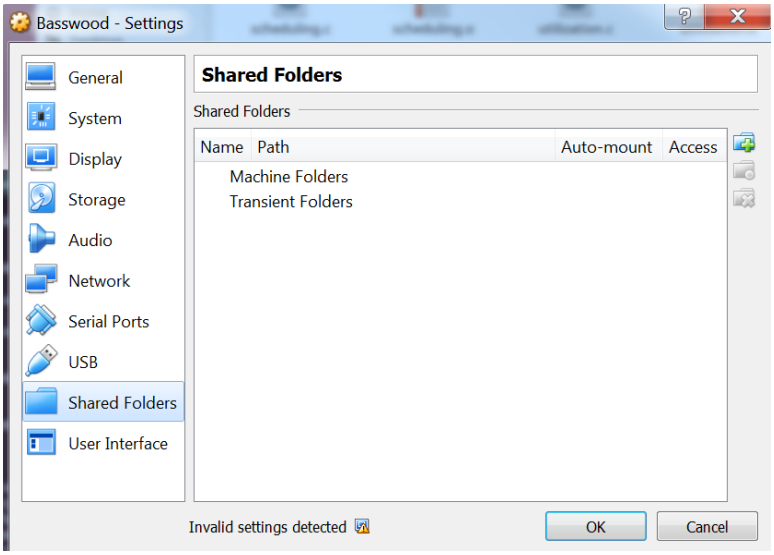


Figure 10. Shared Folder Menu

Create a new shared folder named `basswoodworkspace` with the *Auto-mount* and *Make Permanent* options selected. For Folder Path click the arrow on the right side and navigate to the Basswood folder in your OSATE workspace. Click OK. Verify that the Basswood folder has full access rights and click OK.

From the virtual machine console, create a mount point and mount the share using the following commands, as also shown in Figure 11.

```
$ mkdir /home/basswood/basswoodworkspace
$ sudo mount -t vboxsf -o uid=1000,gid=1000 \
```

```
basswoodworkspace /home/basswood/basswoodworkspace
```

As before, the password is "basswood" in all lower case. Note that Ubuntu does not show any characters while you type your password in the command shell.

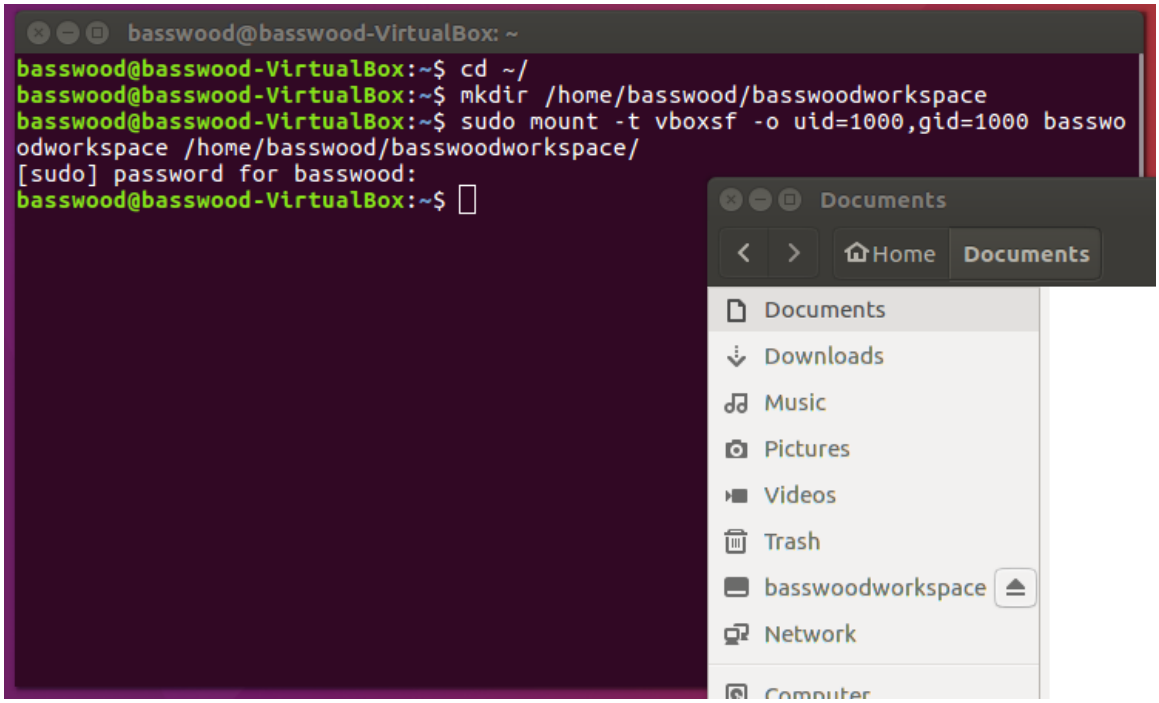


Figure 11. Mounting the Shared Workspace in the Basswood VM

Copy the basswood directory from your host machine share into /home/basswood/development. The exact organization of the host machine share depends on your configuration, but the command will look like the following:

```
$ cp -R /home/basswood/basswoodworkspace/basswood \
   /home/basswood/development
```

Enable execution of the `run` script in `/home/basswood/development/basswood` using the following command:

```
$ chmod +x /home/basswood/development/basswood/run
```

Copy the generated file `scheduling.c` over to the RTEMS environment, under the directory `~/development/basswood/autogen/Basswood`. Replace the existing `scheduling.c` if necessary.

In the Basswood VM execution environment, bring up a terminal and go to the `~/development/basswood` directory (see Figure 12). Build the example FACE-RTEMS system by typing

```
$ make main
```

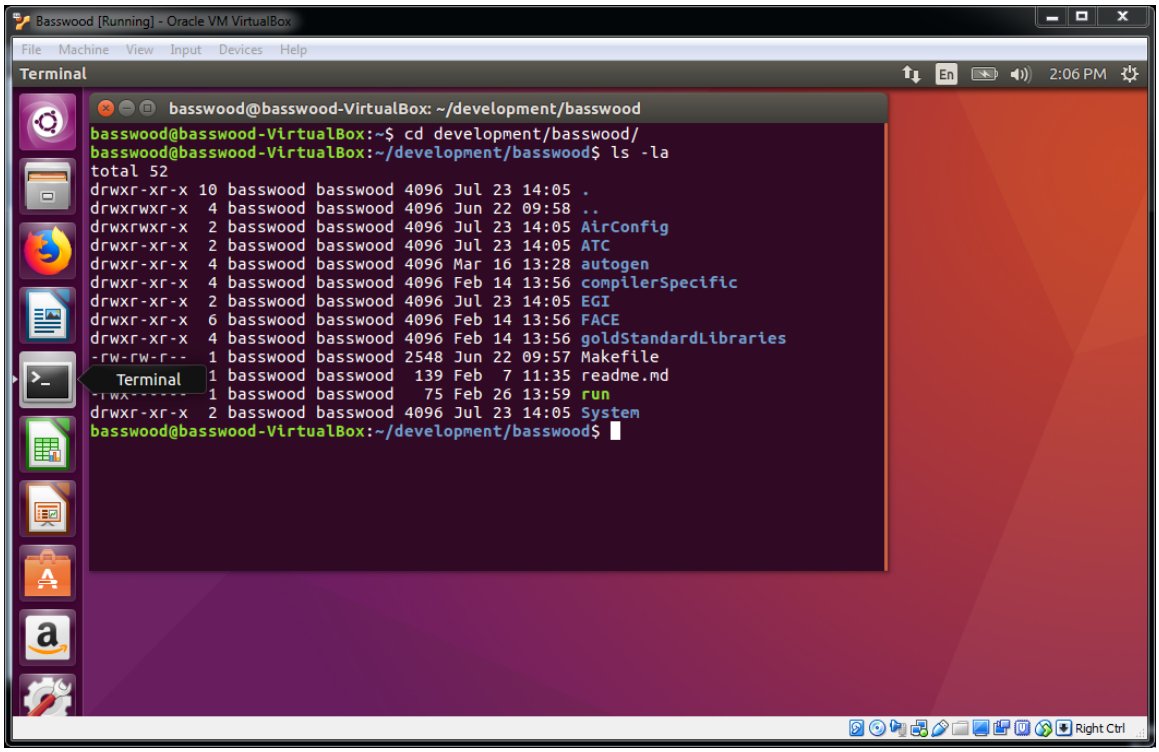


Figure 12. Basswood Build Terminal

When the build is complete, you can invoke a simple script to execute the example model in the RTEMS run-time environment.

```
$ ./run
```

The EGI and AirConfig components will periodically send simulated aircraft and position data to the ATC component. The ATC component receives the data and simply prints out the results as it receives it. The results are streamed to the terminal output. The system is designed to terminate automatically after 10 sets of message iterations between the components, and the number of missed ATC periods is listed at the end. The RTEMS environment will then re-initialize and restart the system. You can stop the run by pressing `Control-c`.

Descriptions of error codes printed by RTEMS can be found in the online RTEMS documentation [https://docs.rtems.org/branches/master/c-user/directive_status_codes.html].

Lesson 4. Utilization Analysis

Prerequisites

- Complete the section called “Lesson 3. Executing a BALSA-Derived Demonstration System” and have the Basswood model in your OSATE workspace
- Complete the section called “Lesson 3. Executing a BALSA-Derived Demonstration System” and have the Basswood demo running on RTEMS
- Install the FASTAR Tool Suite prerequisite tools (see the section called “Additional FACE and AADL Resources”)

Summary

In this lesson, you will learn how to:

1. Add utilization properties to the Basswood AADL model
2. Perform utilization analysis on the Basswood AADL model using the FASTAR tool suite.
3. Demonstrate utilization with Basswood on RTEMS
4. Demonstrate success and failure Basswood on RTEMS
5. Demonstrate utilization failure with the AADL BALSA model

Adding Utilization Properties to the Basswood Model

You learned how to add properties to your AADL models and perform latency analysis using the OSATE tool suite in the section called “Lesson 2. Modifying the BALSA AADL Model”. Now you are going to add FASTAR utilization properties to perform utilization analysis on the Basswood AADL model (see Figure 13) using the FASTAR tool suite installed as a prerequisite for this lesson. Recall from the section called “Introduction to Basswood” that Basswood is a subset of BALSA and is the example you are working with on RTEMS (see Figure 14).

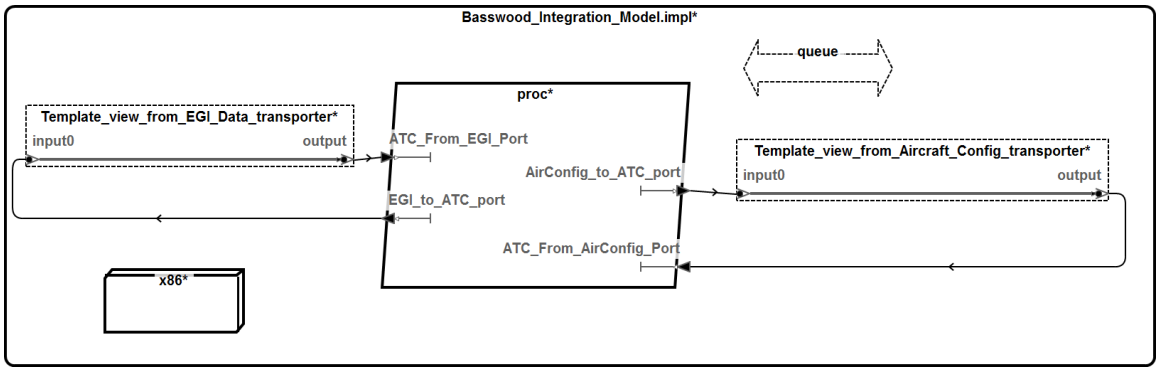


Figure 13. AADL Graphical Model of the RTEMS Basswood Demo Processor, Process, and Communication Infrastructure

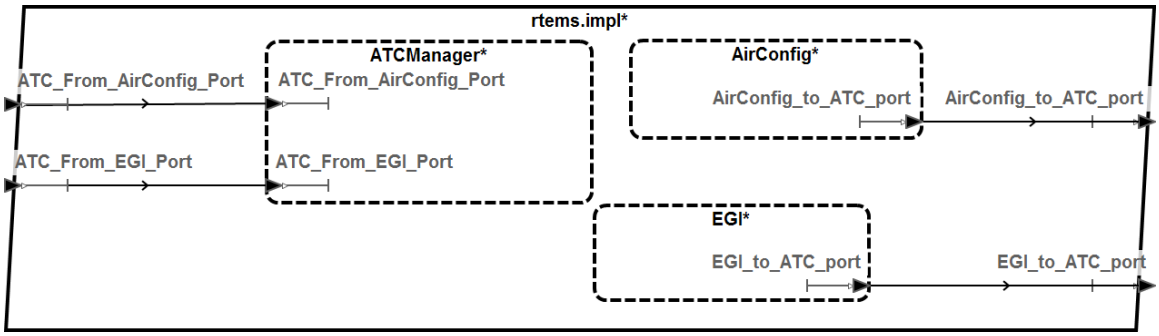


Figure 14. AADL Graphical Model of the Basswood Demo UoP Thread Groups in the rtems.impl Processor

The FASTAR suite includes tools to perform utilization and schedule analysis. Utilization analysis verifies that the resources (e.g. processors, busses, memory) are able to meet the computational, communication, and storage demands of the system (for more information see section 5 of the FASTAR User Guide [https://camet.adventium.com/CAMET/

CAMET/wikis/tool_pages/FASTAR]). Schedulability analysis determines whether the worst case execution times are within the deadlines declared it the model. Verify that the FASTAR property set has been installed with the tool suite by locating it in the `Plugin_Contributions` directory under `Adventium`. To perform FASTAR utilization analysis on the Basswood model, you need to include timing and demand properties and hardware. The Basswood model has (in `basswood_schedule.aadl`) a single thread in each UoP `thread group` in the `rtems.impl` process. Each thread group has a `period` property declaration assigned to it (see Example 30).

```
44  process implementation rtems.impl
45   subcomponents
46    ATCManager: thread group basswood_PCS::ATCManager.impl{
47      period => 100ms;
48    };
49    EGI: thread group basswood_PSSS::EGI.impl{
50     period => 200ms;
51    };
52    AirConfig: thread group basswood_PSSS::AirConfigUoP.impl{
53      period => 1000ms;
54    };

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_schedule.aadl]
```

Example 30. A Sample Thread Execution Period Property

First, add FASTAR processor demand properties to each of the UoP thread groups. Open `basswood_PSSS.aadl` and `basswood_PCS.aadl`. Add a `with FASTAR` declaration to the top of each model. Declare a `MIPS_Demand` of `100.0  MIPS` and a `Code_Size` of `10  Kbyte` for the `AirConfig`, and `EGI` thread groups in `basswood_PSSS.aadl`. Next declare a `MIPS_Demand` of `250.0 MIPS` and a `Code_Size` of `30 Kbyte` for `ATCManager` in `basswood_PCS.aadl`. An example of the property declaration syntax is given in Example 31. `MIPS_Demand` (million instructions per second) is a measure of the demand of a given piece of software on a processor.

```
89   properties
90    FACE::Segment => PSSS;
91    FACE::Profile => safety;
92    FACE::UUID => "_hwTiA0M1EeiBlKadCQCZ8Q";
93    FASTAR::MIPS_Demand => 100.0 MIPS;
94    Code_Size => 10 Kbyte;
95  end EGI;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_PSSS.aadl]
```

Example 31. EGI Thread Group with Resource Demand Properties

In addition to the process `rtems` the `basswood_schedule.aadl` file contains the processor `qemu_x86`. This is the hardware component to which you will be binding your Basswood software model. It has several timing properties laid out in the processor implementation declaration (see Example 32).

```
74  processor implementation qemu_x86.impl
75   properties
76       Clock_Period => 10 ms;
77       FASTAR::Packet_Header_Size =>  100 Bytes .. 100 Bytes;
78       Transmission_Time => [Fixed => 0 ns .. 0 ns; PerByte => 1 us .. 1 us;];
```

```
79  end qemu_x86.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_schedule.aadl]
```

Example 32. Hardware Configuration for the Basswood Model

These properties outline the runtime behavior of the processor. None of these properties are relevant for the utilization analysis portion of the training. You will return to them in the section called “Lesson 6. Schedulability Analysis” for performing schedulability analysis. AADL properties can be added to components in two ways: directly on the component declaration or implementation as shown above, or they can be added in `{ }` brackets when the component is included as a subcomponent of a higher level system. You will be adding the processor supply properties using the latter method to avoid confusing the utilization property declarations with the existing runtime properties used for schedulability analysis.

Navigate to `basswood_integration_model.aadl`. Within `Basswood_Integration_Model.impl` find the processor `x86` subcomponent and add a properties section with `MIPS_Supply` and `Memory_Size` properties as in Example 33.

```
24   system implementation Basswood_Integration_Model.impl
25   subcomponents
26    proc: process basswood_schedule::rtems.impl;
27    x86 : processor basswood_schedule::qemu_x86.impl {
28     FASTAR::MIPS_Supply => 1000.0 MIPS;
29     Memory_Size => 1 MByte;
30     };

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_integration_model.
```

Example 33. Supply and Memory Size Properties for the Basswood System

Now bind the software UoPs to your newly added hardware. A `hardware  binding` is the AADL construct that declares how software logic is partitioned into the available hardware. In the case of Basswood, all of the software is running in the same process on a single processor. Hardware bindings are captured as `Actual_Processor_Binding` and `Actual_Memory_Binding` properties in the top system implementation `Basswood_Integration_Model.impl`with the syntax in Example 34.

```
61    properties
62    Actual_Processor_Binding => (reference (x86)) applies to proc;
63    Actual_Memory_Binding => (reference (x86)) applies to proc;
64    Actual_Connection_Binding => (reference(queue)) applies to connection0,
65     connection1, connection2, connection3;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_integration_model.
```

Example 34. Bind Properties Associating Basswood Software to Hardware

Note that there is already an `Actual_Connection_Binding` property declaration in the model. The connections between system elements are bound to the `queue` virtual bus component, indicating that all of the system communications share a single communication queue.

Performing Utilization Analysis on the Basswood Model

Now that your model has both supply and demand properties, you will perform utilization analysis on the processor. Recall from the section called “Lesson 2. Modifying the BALSA AADL Model” that AADL analy-

sis is performed on instance models. Generate an instance model of your Basswood system by right clicking the `Basswood_Integration_Model.impl` abstract implementation in the **Outline** view and select Instantiate System. Locate your the instance model in the `instances` folder and click on it to select it. Under the Model Analysis menu select Analyze Utilization. The results of your analysis should appear in a new folder named `analysis` and its sub directory named `reports`. The FASTAR utilization analysis tool generates two outputs: XML and CSV. Open the CSV report and verify that your analysis was successful by comparing it to the example shown in Figure 15.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Mode	Supply	Demand	MIPS	MIPS	MIPS	MIPS	MIPS	BPS	BPS	BPS	BPS	BPS	Byte	Byte	Byte	Byte	Byte
2		Resource	Component	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin
3	system.no-mode	Template_view_from																
4	system.no-mode	Template_view_from																
5	system.no-mode	proc.ATCManager																
6	system.no-mode	proc.AirConfig																
7	system.no-mode	proc.EGI																
8	system.no-mode	queue																
9	system.no-mode	x86		1000	450	0.45	1	2.222222						1000000	50000	0.05	1	20
10	system.no-mode	x86	proc															
11	system.no-mode	x86	proc.ATCManager	800	250	0.25	1	3.2						980000	30000	0.03	1	32.66667
12	system.no-mode	x86	proc.ATCManager.ATCThread															
13	system.no-mode	x86	proc.AirConfig	650	100	0.1	1	6.5						960000	10000	0.01	1	96
14	system.no-mode	x86	proc.AirConfig.airConfThread															
15	system.no-mode	x86	proc.EGI	650	100	0.1	1	6.5						960000	10000	0.01	1	96
16	system.no-mode	x86	proc.EGI.egiThread															

Figure 15. FASTAR Utilization Report in Comma Separated Value Format

From the report example in Figure 15, you can see that the processor is able to run Basswood with a `MIPS utilization` of 0.45 and a memory utilization of 0.05 as shown in the highlighted row. Utilization values are proportional; memory utilization of 0.05 means that 5% of available memory is used. These are both feasible values for Basswood running on RTEMS. This is an example of a successful utilization analysis result. In the section called “Demonstrate a Utilization Failure in AADL” you will see an example of a utilization analysis failure. Next you will demonstrate this utilization example using Basswood on RTEMS.

Utilization Success of Basswood on RTEMS

Now you will run this example in the RTEMS execution environment. Start the RTEMS VirtualBox virtual machine (see the section called “Generating Source Code and Running the Real-Time Application” for instructions). Bring up a command line terminal and change the directory to `~/development/basswood/autogen/Basswood`, and start an editor on the source file `utilization.c`.

```
$ vi utilization.c
```

The contents of the file will look something like the following:

```
#include <basswood.h>

rtems_interval egi_demand = 0;
rtems_interval atc_demand = 0;
rtems_interval airconfig_demand = 0;
```

These values represent the time (in milliseconds) per iteration consumed by the EGI, ATC, and AirConfig components respectively. These values are independent of the `period` and `deadline` properties defined on these components, so a system architect can configure the system so that one or more of the components consume more time in their execution than is budgeted, and as a result deadlines are missed.

You will begin, however, with an example that achieves successful utilization for the three example components in the system, EGI, ATC, and AirConfig. Successful utilization means that all three components execute completely without missing their deadlines. In the editor, set the processing demand value to the following:

```
rtems_interval egi_demand = 1;
rtems_interval atc_demand = 10;
rtems_interval airconfig_demand = 1;
```

This defines the execution times for the EGI, ATC, and AirConfig components to be 1 ms, 10 ms, and 1 ms respectively. If the value of the demand is zero, than the component will instead use its period length to represent its demand, effectively assigning the component 100 percent utilization.

You will have to rebuild the system once the changes are made. Save your changes to `utilization.c` and exit the editor (`:wq`). Then go to the `~/development/basswood` directory. Build the example FACE-RTEMS system (see Figure 12) by typing

```
$ make main
```

When the build is complete, you can invoke the `run` script to execute the example model in the RTEMS run-time environment.

```
$ ./run
```

If everything is set correctly, the example will complete 10 iterations without missing a deadline.

Utilization Failure of Basswood on RTEMS

In this section we will reconfigure the example so that the utilization is over-budgeted for one of the threads and as a result deadlines are missed. In the RTEMS VirtualBox virtual machine, return to the directory where the `utilization.c` resides, `~/development/basswood/autogen/Basswood`, and open the file in an editor again. Then change the value of the AirConfig demand to be 90 ms.

```
rtems_interval egi_demand = 1;
rtems_interval atc_demand = 10;
rtems_interval airconfig_demand = 90;
```

This change will force the utilization of the AirConfig thread to a percentage that causes ATC thread to miss deadlines. Rebuild the system and invoke the `run` script to demonstrate a missed deadline.

Demonstrate a Utilization Failure in AADL

Now you will use your Basswood AADL model to recreate a utilization failure similar to what you observed on RTEMS. Navigate to your `MIPS_Demand` property declarations on `ATCManager`, `AirConfig`, and `EGI` and change them to all to `350 MIPS`. Changes to the AADL model made after instantiation are not reflected in existing instance models or analysis results, so you will have to reinstantiate `Basswood_Integration_Model.impl` by right clicking the instance model and selecting the option Reinstantiate. Make sure that your utilization analysis output files are closed and go to Model Analysis>Analyze Utilization to update the output files with the new `MIPS demand` values. The tool will generate a model error at the `Actual_Processor_Binding` property declaration indicating that the processor is over-utilized (see Figure 16).

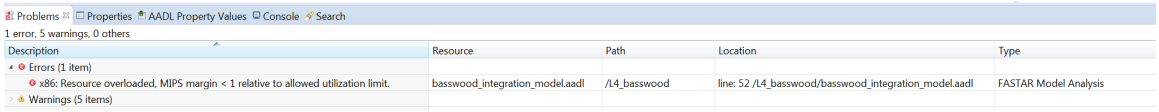


Figure 16. Error Report

Open the CSV file to verify that the processor is over-utilized. The processor in this configuration has a utilization of 1.05 indicating that the processor is indeed overloaded by this configuration as shown in Figure 17 in the highlighted row.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Mode	Supply	Demand	MIPS	MIPS	MIPS	MIPS	MIPS	BPS	BPS	BPS	BPS	BPS	Byte	Byte	Byte	Byte	Byte
2		Resource	Component	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin
3	system.n	Template_view_from_																
4	system.n	Template_view_from_																
5	system.n	proc.ATCManager																
6	system.n	proc.AirConfig																
7	system.n	proc.EGI																
8	system.n	queue																
9	system.n	x86		1000	1050	1.05	1	0.952381						1000000	50000	0.05	1	20
10	system.n	x86	proc															
11	system.n	x86	proc.ATCManager	300	350	0.35	1	0.857143						980000	30000	0.03	1	32.66667
12	system.n	x86	proc.ATCManager.ATCThread															
13	system.n	x86	proc.AirConfig	300	350	0.35	1	0.857143						960000	10000	0.01	1	96
14	system.n	x86	proc.AirConfig.airConfThread															
15	system.n	x86	proc.EGI	300	350	0.35	1	0.857143						960000	10000	0.01	1	96
16	system.n	x86	proc.EGI.egiThread															

Figure 17. Utilization Failure

Now that you have demonstrated the utilization failure case, go back into `basswood_PSSS.aadl` and `basswood_PCS.aadl` and change the `MIPS_Demand` properties back to their initializer values from the section called “Adding Utilization Properties to the Basswood Model”. Reinstantiate the model and perform Utilization analysis once more to remove the warning generated by the failure case.

Lesson 5. Report Generation

Prerequisites

- Complete the section called “Lesson 4. Utilization Analysis” and have the AADL Basswood model with utilization properties in your OSATE workspace
- Download and install the prerequisite model archive in your OSATE workspace (see the section called “Setup”)
- For optional content: Have the CVIS prerequisite tools installed (see the section called “Additional FACE and AADL Resources”)

Summary

In this lesson, you will learn how to:

1. Generate Example Templates
2. Generate a report of your FASTAR utilization analysis
3. Optional: Automate analysis and report generation tasks using Ant

Creating an Example Report Template

The FASTAR Utilization analysis plugin generates two different output files: CSV and XML. The FASTAR tool suite includes a report generation tool that takes the XML output from any analysis tool and builds a format-

ted report using a user-specified template. Example templates can be created for HTML and CSV output reports. Switch OSATE to the XML perspective by selecting the Open Perspective icon in the top right of the OSATE window. Select XML and click Open. Your workspace view on the left side of the screen should now be called Project Explorer. Navigate to the `.xml` output file from the utilization analysis you performed in the section called “Lesson 4. Utilization Analysis”. It is located in the analysis directory in your Basswood project. Right click `Basswood_Integration_Model_impl_utilization.xml` and select the option Create Report Templates. This will create a new subdirectory under `analysis` named `templates`. Within this subdirectory is a folder named `fastar_utilization` which contains two template files. These are example templates for formatting the results of your utilization analysis. For more information on template-based report generation see section 4 of the FASTAR User Guide [https://camet.adventium.com/CAMET/CAMET/wikis/tool_pages/FASTAR].

Generating a Formatted FASTAR Utilization Report

Once you have verified that the templates were successfully created, return to `Basswood_Integration_Model_impl_utilization.xml` and right click to select the option Generate Reports. This will generate a report for each template in the `templates` directory. Formatted reports are found in the `reports` subdirectory of `analysis`. The report generator generated a CSV (see Figure 19) and HTML (see Figure 18) report of your utilization analysis results. Open both reports and verify that they were successfully populated. Scroll down to the table titled *x86* to find your analysis results in the HTML report.

x86

Supply MIPS	Supply BPS	Supply Byte	Demand MIPS	Demand BPS	Demand Byte	Utilization MIPS	Utilization BPS	Utilization Byte	Utilization Limit MIPS	Utilization Limit BPS	Utilization Limit Byte	Margin MIPS	Margin BPS	Margin Byte
1000.0		1000000.0	450.0		50000.0	0.45		0.05	1.0		1.0	2.2222222222222223		20.0

Figure 18. Example HTML Formatted Utilization Report

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Mode	Supply	Demand	MIPS	MIPS	MIPS	MIPS	MIPS	BPS	BPS	BPS	BPS	BPS	Byte	Byte	Byte	Byte	Byte
2		Resource	Component	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin	Supply	Demand	Util	U Limit	Margin
3	system.n	Template_view_from_																
4	system.n	Template_view_from_																
5	system.n	proc.ATCManager																
6	system.n	proc.AirConfig																
7	system.n	proc.EGI																
8	system.n	queue																
9	system.n	x86		1000	450	0.45	1	2.222222						1000000	50000	0.05	1	20
10	system.n	x86	proc															
11	system.n	x86	proc.ATCManager	800	250	0.25	1	3.2						980000	30000	0.03	1	32.66667
12	system.n	x86	proc.ATCManager.ATCThread															
13	system.n	x86	proc.AirConfig	650	100	0.1	1	6.5						960000	10000	0.01	1	96
14	system.n	x86	proc.AirConfig.airConfThread															
15	system.n	x86	proc.EGI	650	100	0.1	1	6.5						960000	10000	0.01	1	96
16	system.n	x86	proc.EGI.egiThread															

Figure 19. Example CSV Formatted Utilization Report

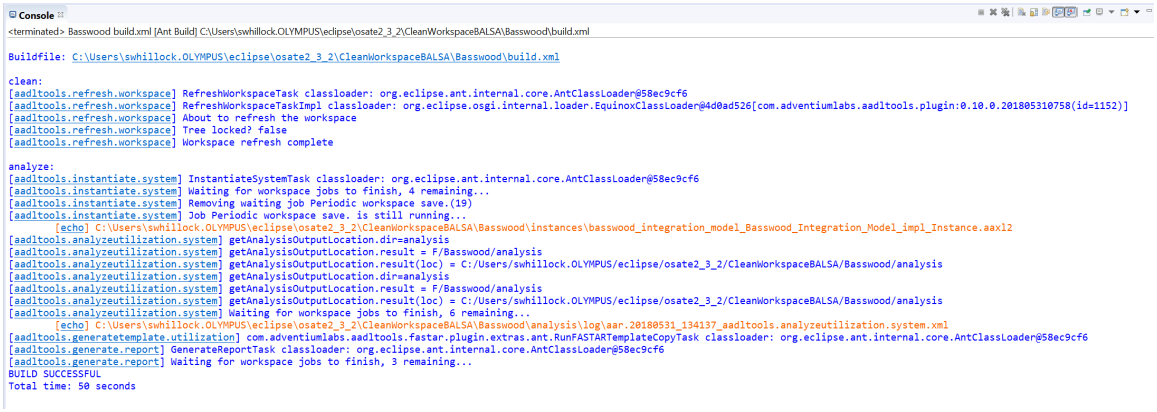
Optional: Automating Analysis and Report Generation using Ant

Ant allows a user to automate a workflow using a script called a `build.xml` file that calls the Ant tasks that constitute the workflow. To create your own Ant workflow for performing utilization analysis and generating a formatted report, navigate to your Basswood AADL project. Verify that you are in the XML perspective by selecting the Open Perspective icon in the top right of the OSATE window. Select XML and click Open. Create a `build.xml` file by going to File>New>XML File. Name it `build.xml` and click Finish. There is an example Ant build script in the `Report_Gen` project that you can look at for guidance (see Example 35).

```
<?xml version="1.0" encoding="UTF-8"?>
 <!-- Must run in the same JRE as the workspace. Set via run -> run as Ant Build... -->
  <project name="Build_Example" default="analyze" basedir=".">

 <!-- This task will clean and refresh the workspace-->
    <target name="clean">
      <delete failonerror="false">
        <fileset dir="analysis" includes="**/*"/>
        <fileset dir="instances" includes="**/*"/>
      </delete>
      <aadltools.refresh.workspace/>
    </target>

  <!-- This task will instantiate the model declared in the modelResource" input -->
    <target name="analyze" depends="clean">
      <aadltools.instantiate.system
       modelResource="Basswood/basswood_integration_model.aadl"
       systemImplName="Basswood_Integration_Model.impl"
       prefix="newinstance"/>
      <echo>${newinstance.instanceFile}</echo>

 <!-- This task will perform FASTAR utilization analysis on the instance -->
      <aadltools.analyzeutilization.system
        instancefile="${newinstance.instanceFile}"
        prefix="basswood_utilization"/>
        <echo>${basswood_utilization.aarFile}</echo>

<!-- This task will generate the example templates -->
        <aadltools.generatetemplate.utilization
        outputProject="Basswood"/>

<!-- This task will generate the formatted report -->
        <aadltools.generate.report
        analysisXMLFile="${basswood_utilization.aarFile}"
        templateResource="Basswood/analysis/templates/fastar_utilization/html_example.html.template"
        prefix="basswood_utilization_reportgen"/>
    </target>
    </project>
```

Example 35. Sample Ant build script

The FASTAR suite includes a set of Ant tasks for automating the steps of performing FASTAR analysis. The list of available tasks can be found in section 4.1 of the CVIS User Guide [https://camet.adventium.com/CAMET/CAMET/wikis/tool_pages/continuous-virtual-integration]. The steps to include in your workflow are the same as those that you went through in the section called “Lesson 4. Utilization Analysis” and the section called “Lesson 5. Report Generation”; create an instance, run utilization analysis on the instance, and generate formatted reports. The example `build.xml` script shown in Example 35 is split into two sections called targets. The first target, **clean**, removes the `instance` and `analysis` directories and refreshes your Eclipse workspace. The second target, **analysis**, generates an instance model, performs FASTAR utilization analysis on that instance, generates the example templates, and creates the HTML and CSV reports. Tasks are called by name and each task has different requirements for user declared inputs and outputs. Check section 4.1 of the CVIS user guide for the requirements of each Ant task. Copy the full example build script and paste it into your new `build.xml` file. Verfy that the file and directory names in each Ant task are correct for your Basswood project. Run the script by right clicking the `build.xml` and select Run As>Ant Build.... Make sure to select the option Run in the same JRE as workspace under the JRE tab of the menu window before clicking Run. The **Console** view at the bottom of your Eclipse window will display the progress of your Ant build. If the **Console** is not open, go to Window>Show View>Console to open it. A successful Ant build will return "BUILD SUCCESSFUL" at the end of the console output (see Example 36).



Example 36. Sample Ant Run Console Output

Lesson 6. Schedulability Analysis

Prerequisites

- Complete the section called “Lesson 4. Utilization Analysis” and have your AADL Basswood model with utilization properties in your OSATE workspace
- Complete the section called “Lesson 3. Executing a BALSA-Derived Demonstration System” and have the Basswood training example running on RTEMS
- Install the FASTAR Tool Suite prerequisite tools (see the section called “Additional FACE and AADL Resources”)

Summary

In this lesson, you will learn how to:

1. Add thread WCET details to the Basswood AADL model
2. Add schedulability analysis properties to the Basswood AADL model
3. Run FASTAR scheduleability analysis on the Basswood AADL model
4. Demonstrate a schedulability failure on the Basswood AADL model
5. Recreate the scheduling failure using Basswood on RTEMS

Adding Timing Properties to the AADL Basswood Model

Switch back to the **AADL Perspective**. Click the Open Perspective icon in the top right of the OSATE window. Select AADL and click Open.Return to the Basswood AADL model you used in the section called “Lesson 4. Utilization Analysis” and open `basswood_schedule.aadl`. Find the thread group property declarations in the `subcomponents` of `rtems.impl`. Each thread group already has a `period` property declaration. Edit the periods so each thread has a period of `1000 milliseconds`. Add `deadline` properties to each of the thread groups that are equal to their periods. Assign a `priority` to each thread group such that `ATCManager` thread group has the highest priority and `AirConfig` has the lowest priority. The FASTAR MAST analysis tool follows the ARINC653 definition of priorities where higher numbers indicate higher priorities. Finally, assign a `200 ms Compute_Execution_Time`

property to each thread group as shown in Example 37. Note that you are using properties on the thread groups to describe the behavior of the thread within it. To clarify to the MAST analysis tool that the properties refer to a particular thread, add an `applies to` clause after each property declaration as shown in Example 37. These property declarations will now override any that are applied to the threads themselves (e.g. the periods that were generated when the Basswood model was initially translated).

```
44  process implementation rtems.impl
45   subcomponents
46    ATCManager: thread group basswood_PCS::ATCManager.impl{
47      priority => 10 applies to ATCThread;
48      period => 1000ms applies to ATCThread;
49      deadline => 1000ms applies to ATCThread;
50      Compute_Execution_Time => 200ms .. 200ms applies to ATCThread;
51    };
52    EGI: thread group basswood_PSSS::EGI.impl{
53     priority => 5 applies to EGIThread;
54     period => 1000ms applies to EGIThread;
55     deadline => 1000ms applies to EGIThread;
56     Compute_Execution_Time => 200ms .. 200ms applies to egiThread;
57    };
58    AirConfig: thread group basswood_PSSS::AirConfigUoP.impl{
59      priority => 1 applies to airConfThread;
60      period => 1000ms applies to airConfThread;
61      deadline =>1000ms applies to airConfThread;
62      Compute_Execution_Time => 200ms .. 200ms applies to airConfThread;
63    };
```

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_schedule.aadl]
```

Example 37. Update period and add deadline and execution times

You have now configured your Basswood system with an execution schedule for the `qemu_x86` processor. This schedule configuration is going to serve as your successful shedulability analysis example. Later you will adjust these values to show what happens when you configure your schedule incorrectly and overload the processor.

Navigate back to `qemu_x86.impl` in the file `basswood_schedule.aadl`. Recall from the section called "Adding Utilization Properties to the Basswood Model", that the processor already has some scheduling properties declared (see Example 38).

```
74  processor implementation qemu_x86.impl
75   properties
76       Clock_Period => 10 ms;
77       FASTAR::Packet_Header_Size =>  100 Bytes .. 100 Bytes;
78       Transmission_Time => [Fixed => 0 ns .. 0 ns; PerByte => 1 us .. 1 us;];
79  end qemu_x86.impl;
```

```
From [/cygdrive/c/Repos/DO3/BAT/training_materials/L4_basswood/basswood_schedule.aadl]
```

Example 38. Existing scheduling properties

The `Clock_Period` property establishes the time it takes the processor to perform any given instruction. In the case of Basswood, the Qemu processor has a clock period of 10 ms. The `Packet_Header_Size` property states that

the processor adds a 100 Byte header to any data packet it processes. The final property is `Transmission_Time`, which states that communications are transmitted at a speed of 1 microsecond per Byte. Add two additional scheduling property declarations to this processor: `Scheduling_Protocol` and `Priority_Range`. `Priority_Range` declares which thread priority levels are handled by a processor. Give `qemu_x86.impl` a priority range of 1 - 20, which indicates that `qemu_x86.impl` is capable of processing all of the threads and thread groups in the Basswood software model. For `Scheduling_Protocol`, declare that `qemu_x86.impl` is using rate-monotonic scheduling (RMS) as the tasks in Basswood have static priorities (see Example 39).

```
120  processor implementation qemu_x86.impl
121   properties
122    Scheduling_Protocol => (RMS);
123       Priority_Range => 1 .. 20;
124       Clock_Period => 10 ms;
125       FASTAR::Packet_Header_Size =>  100 Bytes .. 100 Bytes;
126       Transmission_Time => [Fixed => 0 ns .. 0 ns; PerByte => 1 us .. 1 us;];
127  end qemu_x86.impl;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_schedule.aadl]
```

Example 39. Add scheduling protocol and priority range

Performing Schedulability Analysis on Basswood

Now that you have added scheduling and worst case execution time (WCET) properties to your AADL Basswood, you can add a `FASTAR_Analysis` property declaration to `Basswood_Integration_Model.impl` in `basswood_integration_model.aadl`. This property declaration tells the FASTAR tool which analysis tool to use on your model. For Basswood, you will use `MAST` analysis. The system connecitons are bound to the virtual bus named `queue`. This declares that all threads share the same queue. Finally, declare that all threads use a `Periodic Dispatch_Protocol` as shown in Example 40.

```
60    properties
61    Actual_Processor_Binding => (reference (x86)) applies to proc;
62    Actual_Memory_Binding => (reference (x86)) applies to proc;
63    Actual_Connection_Binding => (reference(queue)) applies to connection0,
64     connection1, connection2, connection3;
65    Dispatch_Protocol => Periodic applies to proc.AirConfig.airConfThread,
66     proc.ATCManager.ATCThread, proc.EGI.egiThread;
67    FASTAR::Analysis => MAST;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_integration_model.
```

Example 40. Add FASTAR Analysis and Periodic Dispatch Protocol

You now have sufficient detail on your Basswood model to yield analysis results from the FASTAR schedulability tool. Create an instance model of `Basswood_Integration_Model.impl` by right clicking it in the **Outline** view and selecting Instantiate. Locate the instance model in the **AADL Navigator** view and make sure it is selected. Run the FASTAR schedulability analysis tool by going to Model Analysis>Analyze Schedulability. The tool will generate warnings in the **Problems** view indicating that your have some components and connections bound to virtual components with no hardware binding (see Figure 20). These warnings are indicating that your `queue` component is not bound to a hardware resource. This is fine for now, as the tool will still perform the schedulability analysis covered by this training.

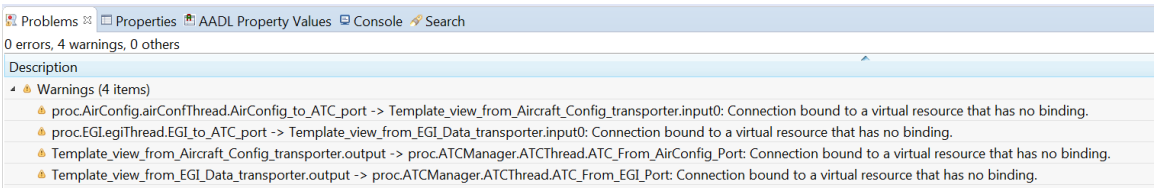


Figure 20. FASTAR warnings about missing hardware bindings

Navigate to the `reports` directory and open the CSV schedulability analysis report (see Figure 21). the section called “Interpreting the Schedulability Analysis Report” explains the contents of the FASTAR schedulability report.

	A	B	C	D	E	F	G	H
1	Mode system.no-mode							
2								
3								
4		BlackBox Resource Analysis						
5			Name	Util	Slack	Q Size		
6			Template_view_from_Aircraft_					
7			Template_view_from_EGI_Dat					
8								
9		BlackBox Task Analysis						
10			Name	Min Latency	Max Latency	Relative to	Jitter	Slack
11								
12		MAST Resource Analysis						
13			Name	Util	Slack	Q Size		
14			proc.ATCManager					
15			proc.AirConfig					
16			proc.EGI					
17			queue					
18			x86	0.599999964				
19								
20		MAST Task Analysis						
21			Name	Min Latency	Max Latency	Relative to	Jitter	Slack
22			proc					
23			proc.AirConfig.airConfThread	200.0ms	600.0ms	proc.AirConfig.airConfThread	400.0000305175781ms	
24			proc.ATCManager.ATCThread	200.0ms	200.0ms	proc.ATCManager.ATCThread	0.0ms	
25			proc.EGI.egiThread	200.0ms	400.0ms	proc.EGI.egiThread	200.0ms	
26								
27		FASTAR Flow Analysis						
28			Flow Name	Element	Segment@End	Remain@Start	Sum@End	Allowed
29			etef					
30				egiThread.EGI_to_ATC_port_source	15.0ms	65.0ms	15.0ms	
31				proc.EGI.egiThread.EGI_to_ATC_port -> Template_vi	0.0ms	50.0ms	15.0ms	
32				Template_view_from_EGI_Data_transporter.flow_p	0.0ms	50.0ms	15.0ms	
33				Template_view_from_EGI_Data_transporter.output	0.0ms	50.0ms	15.0ms	
34				ATCThread.ATC_From_EGI_Port_sink		50.0ms	65.0ms	500.0ms
35			etef2					
36				airConfThread.AirConfig_to_ATC_port_source	15.0ms	65.0ms	15.0ms	
37				proc.AirConfig.airConfThread.AirConfig_to_ATC_por	0.0ms	50.0ms	15.0ms	
38				Template_view_from_Aircraft_Config_transporter.fl	0.0ms	50.0ms	15.0ms	
39				Template_view_from_Aircraft_Config_transporter.o	0.0ms	50.0ms	15.0ms	
40				ATCThread.ATC_From_AirConfig_Port_sink		50.0ms	65.0ms	500.0ms

Figure 21. Schedulability report with a Successful Result

Interpreting the Schedulability Analysis Report

The top sections of the report are the MAST and BlackBox resource analysis results. The BlackBox utilization report should be empty as the only components of the system that are bound to a "black box" resource are the transporter segments that do not have any utilization or schedulability properties tied to them. For Basswood in this configuration, the MAST utilization of the resource `x86` is 0.6 or 60% (see Figure 22). Recall that the utilization analysis tool from the section called “Lesson 4. Utilization Analysis” reported a utilization of 0.45 or 45%. The schedulability tool takes into account the dynamic behavior of the system when computing the processor utilization, thus the discrepency in the result from the utilization analysis tool.

12		MAST Resource Analysis		
13		Name	Util	Slack
14		proc.ATCManager		
15		proc.AirConfig		
16		proc.EGI		
17		queue		
18		x86	0.599999964	

Figure 22. Processor Utilization Report

The next section is the MAST task analysis result that computes worst case execution times for each of the threads against the other tasks and their specified deadlines. Recall that you gave each thread a period and a deadline of 1 second. With three threads, each with a 200 ms execution time, you expect that all three threads will meet their deadlines with the third thread being completed by 600ms. The analysis tool report shows that this is, indeed, the case (see Figure 23).

MAST Task Analysis						
	Name	Min Latency	Max Latency	Relative to	Jitter	Slack
	proc					
	proc.AirConfig.airConfThread	200.0ms	200.0ms	proc.AirConfig.airConfThread	0.0ms	
	proc.ATCManager.ATCThread	200.0ms	600.0ms	proc.ATCManager.ATCThread	400.0000305175781ms	
	proc.EGI.egiThread	200.0ms	400.0ms	proc.EGI.egiThread	200.0ms	

Figure 23. Report Confirming that Deadlines are Met

The final section of the report is the *FASTAR Flow Analysis* (see Figure 24). This is similar to the standard OSATE latency analysis tool you used in the section called “Lesson 2. Modifying the BALSA AADL Model”. The FASTAR flow analysis tool takes into account the dynamic behaviors of the thread executions as well as the `Latency` property declarations. It includes the latency of each segment of the flow in the first column as well as the running total of the flow in the third column. The total flow latency is compared to the declared budget found in the **Allowed** column. Observe that the total flow latency of 65 ms is within the set limit of 500 ms for both end to end flows.

FASTAR Flow Analysis						
	Flow Name	Element	Segment@End	Remain@Start	Sum@End	Allowed
	etef					
		egiThread.EGI_to_ATC_port_source	15.0ms	65.0ms	15.0ms	
		proc.EGI.egiThread.EGI_to_ATC_port -> Template_	0.0ms	50.0ms	15.0ms	
		Template_view_from_EGI_Data_transporter.flow_	0.0ms	50.0ms	15.0ms	
		Template_view_from_EGI_Data_transporter.outpu	0.0ms	50.0ms	15.0ms	
		ATCThread.ATC_From_EGI_Port_sink		50.0ms	65.0ms	500.0ms
	etef2					
		airConfThread.AirConfig_to_ATC_port_source	15.0ms	65.0ms	15.0ms	
		proc.AirConfig.airConfThread.AirConfig_to_ATC_p	0.0ms	50.0ms	15.0ms	
		Template_view_from_Aircraft_Config_transporter	0.0ms	50.0ms	15.0ms	
		Template_view_from_Aircraft_Config_transporter	0.0ms	50.0ms	15.0ms	
		ATCThread.ATC_From_AirConfig_Port_sink		50.0ms	65.0ms	500.0ms

Figure 24. Flow latency within budget

Demonstrating a Schedulability Analysis Failure

Now that you have an example of a schedulability analysis success, you are going to build a schedule that will overload the processor. To avoid overwriting your previous success example, build a second implementation of the `rtems` process in the `basswood_schedule.aadl` file and name it `rtems.fail`. Copy and paste the contents of `rtems.impl` into `rtems.fail` (see Example 41).

```
78  process implementation rtems.fail

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_schedule.aadl]
```

Example 41. Create an implementation that fails

Navigate to the subcomponents of `rtems.fail` and alter the properties of the thread groups as shown in Example 42.

```
81   subcomponents
82    ATCManager: thread group basswood_PCS::ATCManager.impl{
83      priority => 10 applies to ATCThread;
84      period => 100 ms applies to ATCThread;
85      deadline => 100 ms applies to ATCThread;
86
87      Compute_Execution_Time => 50ms .. 50ms applies to ATCThread;
88    };
89    EGI: thread group basswood_PSSS::EGI.impl{
90     priority => 5 applies to EGIThread;
91     period => 200ms applies to EGIThread;
92     deadline => 200ms applies to EGIThread;
93     Compute_Execution_Time => 100ms .. 100ms applies to egiThread;
94    };
95    AirConfig: thread group basswood_PSSS::AirConfigUoP.impl{
96      priority => 1 applies to AirConfThread;
97      period => 1000ms applies to AirConfThread;
98      deadline =>1000ms applies to AirConfThread;
99      Compute_Execution_Time => 100ms .. 100ms applies to airConfThread;
100    };

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_schedule.aadl]
```

Example 42. Alter properties to fail

This configuration should cause the threads to miss their deadlines since the lower priority `AirConfig` thread will be preempted by the higher priority threads when they dispatch at faster rates. In Figure 25, we illustrate the failure scenario you have just set up using your AADL model.

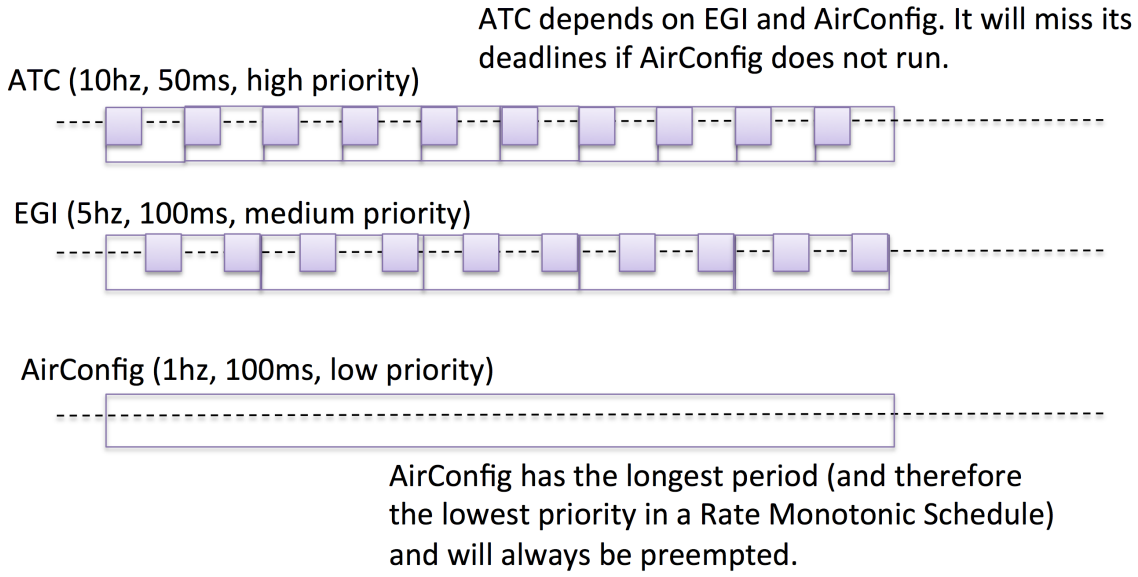


Figure 25. A configuration where threads miss their deadlines

Recall that you declared that the deadlines of each of the threads are equal to their periods. The red line in the diagram indicates the deadline of `AirConfig`.

Next create a copy of `Basswood_Integration_Model.impl` in `basswood_integration_model.aadl` and rename it `Basswood_Integration_Model.fail`. Change the `proc` subcomponent from `rtems.impl` to your newly created implementation `rtems.fail` (see Example 43).

```
72   system implementation Basswood_Integration_Model.fail
73   subcomponents
74    proc: process basswood_schedule::rtems.fail;

From [/cygdrive/c/Repos/DO3/BAT/training_materials/L6_basswood/basswood_integration_model.
```

Example 43. Update to the Proc that will Fail

Generate an instance model of this new system implementation by selecting `Basswood_Integration_Model.fail` in the **Outline** view, right clicking, and selecting the menu option Instantiate. Select the newly generated instance model of `basswood_integration_model.fail` in the **AADL Navigator** view and go to Model Analysis>Analyze Schedulability. The tool will generate the same output report files as earlier, but it will also generate 4 errors on the model as shown in the **Problems** view (see Figure 26).

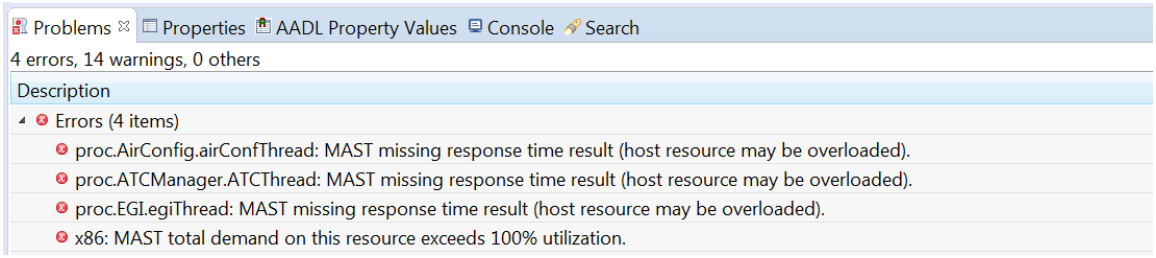


Figure 26. FASTAR Errors

The errors generated by the tool indicate that the threads are missing their deadlines and the processor is overloaded. Open the CSV report and note that it shows the same error outputs as the console (see Figure 27).

	A	B	C
1			
2	Errors		
3		Component	Message
4		x86	MAST total demand on this resource exceeds 100% utilization.
5		proc.AirConfig.airConfThread	MAST missing response time result (host resource may be overloaded).
6		proc.ATCManager.ATCThread	MAST missing response time result (host resource may be overloaded).
7		proc.EGI.egiThread	MAST missing response time result (host resource may be overloaded).

Figure 27. Schedulability report with errors

You have now successfully generated a model of a priority inversion scenario. Next you will demonstrate a priority inversion using Basswood on RTEMS.

Executing the Priority Inversion in RTEMS

You are ready to see the priority inversion example execute in the RTEMS run-time environment. Refer to the section called “Lesson 3. Executing a BALSA-Derived Demonstration System” to recall how to generate source code for the

RTEMS run-time environment and how to configure and start the RTEMS VirtualBox virtual machine. To run the example, perform the following steps:

- Generate the source code from the AADL model, which results in a new `schedule.c` file.
- Start and login to the RTEMS VirtualBox virtual machine, and copy the `schedule.c` to the `~/development/basswood/autogen/Basswood` directory (overwrite the old copy if necessary).
- Bring up a command line terminal, and in the same directory, open the `utilization.c` in an editor. Set each **_demand* to zero. This will set the execution time of each component in the system to equal their worst-case execution time by default.
- Return to the `~/development/basswood` directory, and build the system by typing

```
$ make main
```

- When the build is complete, you can invoke a simple script to execute the example model within the RTEMS run-time environment.

```
$ ./run
```

- The EGI and AirConfig components will periodically send simulated aircraft and position data to the ATC component. The ATC component receives the data and simply prints out the results as it receives it. The results are streamed to the terminal output. The system is designed to terminate automatically after 10 sets of message iterations between the components, and the number of missed ATC periods is listed at the end. The RTEMS environment will then re-initialize and restart the system. You can stop the run by pressing `Control-c`.

