École Polytechnique de Montréal

AER8300: Lab #3

February 20, 2019

Giovanni Beltrame <giovanni.beltrame@polymtl.ca>
Yanjun Cao <yanjun.cao@polymtl.ca>

1 Practical Information

Deliverables	An archive (.zip, .tar, .tar.gz) with your AADL model. You can organize your model in multiple packages across multiple files. A single PDF document in English describing the model (similarly to what is done for the example in the Appendix of these instructions).
Deadline	March 17, 2019 at 23:59 pm.
How to submit	By e-mail at yanjun.cao@polymtl.ca.
Teamwork	The 2-people groups are the same as for Lab #1.
Evaluation	Up to 20% (of 40%). All the members of a group will receive the same evaluation.

2 Set Up

2.1 Java Version

Make sure that you have Java installed. OSATE2 requires a version >=1.8 of the Java VM. You can find different updates of JRE and JDK 8 using these links:

- `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`
- `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`

If you are using Windows, you can check which version(s) of Java you have installed browsing:

```
Start > Control Panel > Programs > Uninstall a Program
```

or

```
Start > CMD > java -version
```

2.2 Get OSATE2

OSATE2 provides independent package for Window, which is integrated with Eclipse. The software is a green package which can run without installation in Windows. Follow `https://osate.org/download-and-install.html#new-installation` and `https://osate.org/download-and-install.html#download-locations` to get the OSATE2 for your computer.

Note that the Help document inside the software is very helpful. Detailed tutorials are provided, which could be hard to find by google sometimes.

3 Tasks

In this lab, you will use the IDE to write the AADL code that describes a complex subsystem of a satellite.

Read the document called `DESCRIPTION.OF.A.SATELLITE.ATTITUDE.AND.ORBIT.CONTROL.SUBSYSTEM.pdf`. It contains the description of a real attitude and orbit control system.

Task 1 (20 points): Create an AADL model of the system described in the document. The more complete the model is, the higher will be your score. Write a PDF report to document it.

4 Hints

- To give you an idea of what the deliverable should look like, an acceptable solution could be composed by four AADL files (of $\sim$150 lines each) describing, respectively: (i) datatypes, (ii) software, (iii) hardware, and (iv) the overall system. Different approaches can be acceptable as well. The evaluation of your work will only be based on completeness and consistency (with respect to the description in the `.pdf` file).
- You can/should consider satellite components that are not internal to the AOCS but you think that are directly connected to your model (e.g., main bus, main computer, telemetry, etc.)
- The purpose of this lab is the introduction of AADL and the scope of the use of OSATE2 is limited to its syntax checker and its graphical interface (you can create diagrams from your model or draw a system for which a model is automatically generated). Some of AADL's advanced functions for model checking and scheduling analysis (implemented by Ocarina) will presented through the TASTE toolchain in one of the following TP session.
- On the next page, you will find a step-by-step introduction to the AADL description of a much simple AOCS that you can use as a starting point for your solution (OSATE2 includes examples too).

Appendix: AADL Concepts
by Carlo Pinciroli

What is AADL?

AADL stands for "Architecture Analysis and Design Language". Its main purpose is to allow a designer to describe the elements of the system that must be constructed, as well as the interactions of these elements.

AADL is more than just a conventional language. After all, any programming language (e.g., Python, Java, etc.) does the same job. The power of AADL stems from the fact that it is an *abstract* language, in the sense that it is capable of describing *both* the hardware and the software aspects of your system. With a conventional programming language you would not be able to do this.

Furthermore, systems described with AADL (also called *AADL models*) are *executable*. This means that once your model is ready, you can press a button and see how it behaves. The ability to *analyze* a design is a crucial feature to spot issues early on during the lifetime of your project, when making mistakes costs little. The later you spot a design issue, the more expensive (in terms of time, money, etc.) it is to fix it.

AADL Components

At first, AADL looks extremely confusing. There are dozens of different names to go through, and none of them seems to find a purpose to exist.

The main point to make sense of this zoo is to understand that systems (i.e., what you want to design) are made of *components*. A satellite, for instance, is composed of hardware parts (e.g., CPU, memory, antennas, sensors, boosters, etc.) and software parts (e.g., for orbital control, scientific data analysis, communication). AADL offers tools that aid you in the definition of each component. These tools are usually called (surprise!) *AADL components*.

AADL offers components for software aspects, such as processes and threads, and components for hardware aspects, such as hard disks, sensors, and network connections. In addition, AADL gives you means to define how the components are connected (i.e., how they interact).

To summarize: the ultimate aim of AADL is to offer a component for any possible aspect you might want to describe in your model.

AADL Workflow

OSATE stands for "Open Source AADL Tool Environment.". Its purpose is to offer you an integrated development environment (IDE) to work on AADL models. We will go through the main phases of defining and analyzing an AADL model in OSATE.

Requirement Specification

We are asked (again!) to design a satellite capable of taking pictures of Montreal every two weeks. As part of the work, we need to design an attitude controller that points the satellite towards the sun, and we want the frequency of the control action to be 10Hz.

Model Definition

The decision on which aspects are relevant for the design of any system depends on the ingenuity and experience of the designer. Often one starts with a simple design, which undergoes progressive refinement as mistakes are discovered or more information becomes available.

4.0.1 Defining Component Interfaces

In the spirit of simple modeling, let us start by defining our satellite as an AADL component. AADL offers several types of components: `data`, `subprogram`, `thread`, `process`, `memory`, `device`, `processor`, `bus`, `system` and many others. Since a satellite is an object made of multiple parts, the right AADL component for the

job is `system`. AADL models can be specified in textual form or in graphical form, but here we will focus on the textual one:

```
system Satellite
end Satellite;
```

This is not a correct model yet, because the AADL syntax imposes that components must be declared within a package. Thus, we wrap our component in a package:

```
package Lab3
public
  system Satellite
  end Satellite;
end Lab3;
```

Let us concentrate now on the elements of the satellite that we want to consider. Since we want to model an attitude controller, we need three essential elements:

1. A sensor that gives us the direction to the Sun;
2. An actuator to rotate the satellite;
3. A piece of software that takes the sensor reading and feeds the actuator with the right values;
4. An onboard computer that runs the attitude controller.

The Sun sensor is a hardware component, so we will capture it as a `device`. Its job is to provide the direction to the Sun expressed as three Euler angles. We model the output of the sensor as three output data ports, one for each Euler angle:

```
package Lab3
public
  system Satellite
  end Satellite;

  device SunSensor
    features
      rot_x: out data port;
      rot_y: out data port;
      rot_z: out data port;
  end SunSensor;
end Lab3;
```

We can say more about this sensor. We know that the output of the sensor is an angle, that is, a floating-point variable. We can add this piece of information to the model by creating a new data type that we call `radians` whose representation is a `float`. As you might have guessed, data types are yet another AADL component:

```
data radians
  properties
    Data_Model::Data_Representation => float;
end data;
```

In order to be able to use the `Data_Model` definition, we need to import it in the package. The script becomes:

```
package Lab3
public
  with Data_Model;

  system Satellite
  end Satellite;

  data radians
  properties
    Data_Model::Data_Representation => float;
  end radians;

  device SunSensor
    features
      rot_x: out data port radians;
      rot_y: out data port radians;
      rot_z: out data port radians;
  end SunSensor;
end Lab3;
```

Next, the torque actuator: analogously to the Sun sensor, we will model it as a `device` and endow it with three input ports, one for each axis of rotation. We also define the data type `torque`:

```
package Lab3
public
  with Data_Model;

  ...

  data torque
    properties
      Data_Model::Data_Representation => float;
  end torque;

  device Torquer
    features
      tor_x: in data port torque;
      tor_y: in data port torque;
      tor_z: in data port torque;
  end Torquer;
end Lab3;
```

Now we add the attitude controller. Being a standalone piece of software, the right component type to model it is `process`. Other software components, such as `thread`, `thread group` or `subprogram`, are used to specify how a program is structured. We stick to a simple model and represent the controller as a component with three inputs (the Euler angles) and three outputs (the torques):

```
package Lab3

  ...

  process Controller
    features
      rot_x: in data port radians;
      rot_y: in data port radians;
      rot_z: in data port radians;
```

```
      tor_x: out data port torque;
      tor_y: out data port torque;
      tor_z: out data port torque;
  end Controller;
end Lab3;
```

The last `device` in our model is the on-board computer, which takes as input the sensor readings and the torques decided by the controller, and outputs the sensor readings (to the controller) and the torques (to the torquer):

```
package Lab3

  ...

  device Computer
    features
      rot_x_in: in data port radians;
      rot_y_in: in data port radians;
      rot_z_in: in data port radians;
      rot_x_out: out data port radians;
      rot_y_out: out data port radians;
      rot_z_out: out data port radians;
      tor_x_in: in data port torque;
      tor_y_in: in data port torque;
      tor_z_in: in data port torque;
      tor_x_out: out data port torque;
      tor_y_out: out data port torque;
      tor_z_out: out data port torque;
  end Computer;
end Lab3;
```

Defining Component Implementations

At this stage, our model is missing lots of details. We only defined the interface of the components, but not *(i)* what the components do, nor *(ii)* how the components are connected. To define these aspects, AADL has the concept of *component implementation*.

Let us start by defining the connections among the components. We defined `Satellite` as a system that contains everything. Its implementation must say *(i)* that the system includes the other components and *(ii)* how the components are connected:

```
package Lab3

  ...

  system implementation Satellite.impl
   subcomponents
   sens: device SunSensor;
   act: device Torquer;
   cpu: device Computer;
   contr: process Controller;
   connections
   SnCpX: port sens.rot_x -> cpu.rot_x_in;
   SnCpY: port sens.rot_y -> cpu.rot_y_in;
   SnCpZ: port sens.rot_z -> cpu.rot_z_in;
   CpCnX: port cpu.rot_x_out -> contr.rot_x;
```

```
   CpCnY: port cpu.rot_y_out -> contr.rot_y;
   CpCnZ: port cpu.rot_z_out -> contr.rot_z;
   CnCpX: port contr.tor_x -> cpu.tor_x_in;
   CnCpY: port contr.tor_y -> cpu.tor_y_in;
   CnCpZ: port contr.tor_z -> cpu.tor_z_in;
   CpAcX: port cpu.tor_x_out -> act.tor_x;
   CpAcY: port cpu.tor_y_out -> act.tor_y;
   CpAcZ: port cpu.tor_z_out -> act.tor_z;
  end Satellite.impl;
end Lab3;
```

We are not done with the connections. In fact, on the real satellite, data would be exchanged using a bus. Let us add a bus to our model. A bus is yet another AADl component:

```
package Lab3

  ...

  bus CommBus
  end CommBus;

end Lab3;
```

and update the definition of our components to say that they need to use the bus:

```
package Lab3

  ...

  device SunSensor
    features
      ...
      comm: requires bus access CommBus;
  end SunSensor;

  device Torquer
    features
      ...
      comm: requires bus access CommBus;
  end Torquer;

  device Computer
    features
      ...
      comm: requires bus access CommBus;
  end Computer;

end Lab3;
```

What remains to do is defining the behavior of each component. The Sun sensor, for instance, takes measurements periodically—let us say 10 times per second:

```
package Lab3

  ...
```

```
  device implementation SunSensor.impl
   properties
   Period => 100ms;
  end SunSensor.impl;

end Lab3;
```

while the actuator and the computer have nothing special to report:

```
package Lab3

  ...

  process implementation Torquer.impl
  end Torquer.impl;

  device implementation Computer.impl
  end Computer.impl;
end Lab3;
```

Finally, let us update `Satellite.impl`:

```
package Lab3

  ...

  system implementation Satellite.impl
   subcomponents
      ...
   comm: bus CommBus;
   connections
      ...
   CommSn: bus access comm <-> sens.comm;
   CommAc: bus access comm <-> act.comm;
   CommCp: bus access comm <-> cpu.comm;
  end Satellite.impl;
end Lab3;
```

For the implementation of the controller, we need to specify how the data is processed. Let us that the data is processed in a thread executed 10 times per second:

```
package Lab3

  ...

  thread ControlThread
    features
      rot_x: in data port radians;
      rot_y: in data port radians;
      rot_z: in data port radians;
      tor_x: out data port torque;
      tor_y: out data port torque;
      tor_z: out data port torque;
  end ControlThread;

  thread implementation ControlThread.impl
```

```
    properties
      Dispatch_Protocol => Periodic;
      Period => 100ms;
  end ControlThread.impl;

  process implementation Controller.impl
    subcomponents
      worker: thread ControlThread.impl;
    connections
      Rx: port rot_x -> worker.rot_x;
      Ry: port rot_y -> worker.rot_y;
      Rz: port rot_z -> worker.rot_z;
      Tx: port worker.tor_x -> tor_x;
      Ty: port worker.tor_y -> tor_y;
      Tz: port worker.tor_z -> tor_z;
  end Controller.impl;

end Lab3;
```

Data Flow

The ultimate goal of specifying a model in AADL is to be able to assess the properties of the system-to-be. Our current model, however, does not allow us to analyze much—after all, we only modeled how the components are connected to each other.

One aspect that can be analyzed easily in AADL is *data flow*. In our example, data flows from the Sun sensor, are received by the onboard computer, processed by the controller thread, and the result is fed to the torque actuator. Data flow in a real system cannot be instantaneous: every component adds a certain latency. In a real-time system it important to verify that such latency is never too high, to prevent the system from missing deadlines with potentially catastrophic consequences. In our case, for instance, we want the system to be able to control attitude 10 times per second (see Section 4). If information flowed slower than that, we would know that something must be fixed in our system.

Data flow in AADL is specified with three concepts: `source` (e.g., the Sun sensor), `path` (e.g., the controller thread) and `sink` (e.g., the torque actuator).

The Sun sensor takes between 1 and 2 milliseconds to generate its data:

```
package Lab3
  ...
  device SunSensor
    features
      ...
    flows
      fx: flow source rot_x {latency => 1ms .. 2ms;};
      fy: flow source rot_y {latency => 1ms .. 2ms;};
      fz: flow source rot_z {latency => 1ms .. 2ms;};
  end SunSensor;
end Lab3;
```

The controller takes 3-5 milliseconds to calculate the outputs:

```
package Lab3
  ...
  process Controller
    features
      ...
    flows
```

```
      fx: flow path rot_x -> tor_x {latency => 3ms .. 5ms;};
      fy: flow path rot_y -> tor_y {latency => 3ms .. 5ms;};
      fz: flow path rot_z -> tor_z {latency => 3ms .. 5ms;};
  end Controller;
end Lab3;
```

The torquer takes between 1 and 2 milliseconds to actuate the control input:

```
package Lab3
  ...
  device Torquer
    features
      ...
    flows
      fx: flow sink tor_x {latency => 1ms .. 2ms;};
      fy: flow sink tor_y {latency => 1ms .. 2ms;};
      fz: flow sink tor_z {latency => 1ms .. 2ms;};
  end Torquer;
end Lab3;
```

Documentation and Examples

For further information, have a look at these links:

- http://www.openaadl.org/post/2013/04/15/aadl-tutorial/
- http://www.openaadl.org/post/2014/09/28/models/
- http://www.aadl.info/aadl/documents/OverviewSAEAADL-June_2009.pdf
- http://ptgmedia.pearsoncmg.com/images/9780321888945/samplepages/0321888944.pdf