Learn AADL concepts in a pleasant way

Julien DELANGE, Peter FEILER
at Software Engineer Institute, Carnegie Mellon University
delange@enst.fr, phf@sei.cmu.edu

February 24, 2009
Contents

1 Before you start 3
  1.1 Introduction ................................................. 3
  1.2 Requirements ................................................ 3
    1.2.1 Required background .................................... 4
    1.2.2 Get Help ! ................................................ 4
    1.2.3 Goal of this document .................................... 4
    1.2.4 Improve this document ................................. 5
  1.3 Using the Ocarina plugin in Eclipse/Topcased .............. 5
    1.3.1 Configure the plugin ..................................... 5

2 Understand the basic of AADL 6
  2.1 Hello World (first exercise) ............................. 6
    2.1.1 What is missing in this model ? ......................... 7
    2.1.2 What should be added and how to add it ? .............. 8
    2.1.3 Adding missing informations ............................ 8
    2.1.4 Test our implementation ! ............................... 9
    2.1.5 Understand what is created ............................. 9
    2.1.6 What could be expected now ? .......................... 9
  2.2 Network communications with AADL (exercise 2) .......... 9
    2.2.1 What is missing in my model ? ......................... 10
    2.2.2 What should I add ? ..................................... 10
    2.2.3 Test and run your system ............................... 11
  2.3 Understand the semantic of the process component (exercise 3) ................................. 11
    2.3.1 Test and run your system ............................... 12
    2.3.2 Understand the changes ................................. 13

3 Advanced use of AADL 14
  3.1 Streaming music system ................................. 14
    3.1.1 What is a streaming server ? ......................... 14
    3.1.2 Why this example illustrates real-time principles ? .... 14
  3.2 Understanding delays and jitter (exercise 4) .............. 15
    3.2.1 Understand our first streaming example ............... 15
    3.2.2 Understand the impact of Periods ....................... 16
    3.2.3 Understand the impact of latency and jitter ............ 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4</td>
<td>Understand the difference between time-driven and event-driven tasks</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Understanding faults and their impact</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Understanding our second streaming example</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Test and run your system</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Hear impact of faults (exercise 5)</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Thread period</td>
<td>19</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Execution time of a thread, simulate jitter issues</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>Hear impact of faults on your system</td>
<td>20</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Add a random fault to your communication</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusion</td>
<td>21</td>
</tr>
</tbody>
</table>
Chapter 1

Before you start

1.1 Introduction

This document is a tutorial to learn AADL, its concept and its semantics. Each new language is difficult to learn, it introduces new concepts that are sometimes difficult to understand. Moreover, Model Based Engineering (MBE) could be seen as abstract. We explain how MBE can help system designers and developers in their work.

This tutorial is made for people that want to enjoy learning the AADL. We describe the basics of the language and explain them through concrete examples. Moreover, for most concepts of AADL, we automatically generate the code and generate programs that represent your model.

For interested readers, a bibliography is available at the end of this document. It gives you some pointers to learn more about the AADL and related work.

1.2 Requirements

This tutorial requires the following software installed on your system:

- Topcased or Eclipse with the OSATE plugin (see http://www.topcased.org)
- Ocarina and its plugin for Topcased or Eclipse/OSATE (see http://aadl.enst.fr)
- The SDL library\(^1\)

Topcased is a toolsuite to model safety-critical systems. It is based on Eclipse and provides facilities to use several modeling languages. It can be retrieved on http://www.topcased.org and is available on most development platforms (Linux, Mac OS X and Windows). It also works on most UNIX-based systems (such as FreeBSD) that support Linux emulation.

\(^1\)http://www.libsdl.org
Ocarina[3] is an AADL compiler that analyzes AADL models and generate Ada or C code. It has also other features but we will explore all its functionalities. You can download it on http://aadl.enst.fr (Ocarina snapshots).

The SDL library is a cross-platform library to create interoperable multimedia systems (games, ...). It is also used in commercial products. In our context, we use it to play music. You must have the compiled shared library and development headers.

However, people that expect a more traditional way to learn the AADL can refer to the AADL standard[1] or to its introduction[2].

1.2.1 Required background

We assume that you know how to create AADL models inside Topcased. You will also give some hints to create AADL models inside Topcased but we assume that you are conformable with a such graphical interface.

1.2.2 Get Help!

You’ll probably be lost when you will try to make this tutorial. It’s normal and many tutorials don’t explain enough deeply some concept or forget to notice important things that seem not relevant for the writer but are very important for the reader. Video/screenscasts that show the exercises are available at our AADL portal2. We also provide screenscast on the archive. You can also send us an email when something seem unclear to you.

1.2.3 Goal of this document

The role of this document is to easily understand AADL concepts and how we can implement systems using this language. We emphasize on the language, because it is closed to the implementation and facilitates the mapping to code.

AADL is not just a modeling language, it really helps you to create implementations. In this tutorial, we will see the following example:

- One process with two threads
- Two processes that exchange data through the network
- A streaming system: two processes that music data through the network
- Understand the semantic of the process component
- Hear and understand impact of latency, jitter and scheduling requirements.
- Hear and understand impact of fault in your system.

Each example will be described and we will explain how to model these systems in AADL and how we implement them.

---

2http://aadl.enst.fr
1.2.4 Improve this document

This document can be improved in many ways. If you find any way of improvement, you can send your comments to the author. This tutorial was written using Linux, you could encounter some difficulties on other platforms. In this case, please contact the author.

1.3 Using the Ocarina plugin in Eclipse/Topcased

The Ocarina plugin work on Eclipse/Topcased. This is an interface between Eclipse and Ocarina AADL toolsuite. You can download it on http://aadl.enst.fr/ocarina/eclipse/. Before using it, you have to configure it. Go to your preferences, there is a dedicated entry for configuring Ocarina. If you experience problems using our plugin, you can get help on the AADL portal at ENST\(^3\).

1.3.1 Configure the plugin

The configuration information are used to run Ocarina on top of Eclipse. At first, you must provide the directory where Ocarina is installed. This is the directory where the file ocara or ocara.exe resides.

If your model uses AADL version 2, please check the appropriate box.

Finally, Ocarina can generate code for several languages. Here, we will use the generator PolyORB-HI-C. It generates code compliant with the POSIX standard.

\(^3\)http://aadl.enst.fr
Chapter 2

Understand the basic of AADL

2.1 Hello World (first exercise)

In this first example/exercise, we will design a system with two tasks. These tasks will simply print some text like "Hello, i’m task one !" or "Hello, i’m task two!". A such exercise is a good to understand the basis of AADL: how to model system’s architecture, describe properties and requirements of each part of the system.

At first, create a new AADL project in Topcased and add a new AADL model in this project. The name of the file that contains the AADL model must be hello.aadl. This model already contains components, we will explain what is the sense of each component.

This model contains one system named hello.impl. This is the root system of our AADL model. It means that this is the topmost component in our system, the root one that contains all other relevant component. Each AADL model has a such root component. It must be a system component.

In this hello.impl system, there are subcomponents: a process and a processor. There is also some properties, we will describe them further. In AADL, each component can have subcomponents and properties. subcomponents define what is contained in the current component (for example, a process can contain threads, a thread can contain data and so on). properties describe the properties and the requirements of the component and its subcomponents. It could be added on each AADL component. For example, the properties can describe what is the size of a data, what is the scheduling algorithm for the system and so on. Let’s return to our AADL model, we will explain the meaning of each subcomponents of our system hello.impl.

The process component has the same meaning than a UNIX process. It is a separate address space that contains code and data. It contains threads to execute code.

The processor component models the hardware component called processor but also the underlying runtime (meaning, the operating system). The concept of operating system can be seen as similar to the concept of underlying runtime: it is a set of functionalities to create and manage the resources of your system (processes, threads, data, locking protocols and so on).
So, our system contains a process and a processor. Model a single process without its execution runtime is useless. Model a processor without entities to execute is useless too. Consequently, we have to associate components and indicate which component is bound to another.

That’s why we define the property Actual_Processor_Binding. This property indicates that the cpu_rm component is bound to the node_a process. So, this process will use the operating system we model through the cpu_rm component.

So now, we know that our system contains one process bounded to one processor. Now, we will explore the process component.

The process component contains two threads. It means that this address space contains two threads that execute code. It is the same mechanism than with the system component. Let now explore the thread component.

The thread components have a new sections called calls. This section describe what subprograms are executed by the thread. In other word, this section models executed code. Indeed, we model thread but we didn’t point out what is executed. The call sequence defines precisely what subprograms are called and the order of calls.

We can also see at lot of properties added to the thread component. The Dispatch_Protocol specify what kind of thread we have. We distinguish two kind of threads:

- **Periodic threads** executes the same call sequences of subprograms at a certain rate (we also call that the Period).

- **Sporadic threads** execute a call sequence when an event is raised. The call sequence may depend on the event.

We will explain further the internals of sporadic threads. The Period property specify the rate at which the task is repeated. The Compute_Execution_Time specify the time needed to complete the execution of the call sequence. Finally, the Deadline property specify when the task should complete its work.

The subprogram components model C functions, Ada procedures or functions, etc. Subprogram components are called by threads.

Now, we have a complete architecture model: a system that contains a processor and a process. The process contains two threads and each thread calls a subprogram. Look easy, no ? Note that our model make less than 100 lines. This is why AADL is very efficient: you can easily describe your system without having an heavy model.

### 2.1.1 What is missing in this model ?

We model the architecture of our system, describing its components. However, it is useless until we implement it. To do so, we will generate code for our system. However, several things miss. We don’t specify on which operating system we want to run our system. It could be useful to specify the system in order to create a code dedicated to this system.

Moreover, we specify the architecture of our system, but we don’t wrote the functions ! The model does not indicate what operations are performed by the subprograms. To do so, we have to write the functions. Moreover, we must indicate to which C function a subprogram correspond.
2.1.2 What should be added and how to add it?

We must add a bunch of properties to each subprogram. There is the missing properties on each subprogram:

- **Source_Language**: this property specifies what is the language used to implement a subprogram.
- **Source_Name**: indicates what is the name of the function that implements this subprogram.
- **Source_Text**: indicates in which files the functions resides.

More, we must write the functions that implements our subprograms. We will do that in the next subsection.

In addition, we must indicate what his the underlying architecture and which operating system will be used. The processor component represents the operating system, so we have to add a property to this component.

2.1.3 Adding missing informations

You have to complete your model with the missing properties. After modifications, the subprogram components should look like this:

```
subprogram Hello_Spg_1
properties
  source_language => C;
  source_name => "user_Hello_Spg_1";
  source_text => ("hello.c");
end Hello_Spg_1;

subprogram Hello_Spg_2
properties
  source_language => C;
  source_name => "user_Hello_Spg_2";
  source_text => ("hello.c");
end Hello_Spg_2;
```

This tell the code generator that the subprogram `Hello_Spg_1` is implemented in a function written in the C language and called `user_Hello_Spg_1`. It is located in a file called `hello.c`, in the same directory than the model.

Finally, create a C file in the same directory than your models and write the functions `user_Hello_Spg_1` and `user_Hello_Spg_2`. Be careful, the name of the file must be the same than the value of the `Source_Text` property. The file should look like this:

```
#include <stdio.h>

void user_hello_spg_1 (void)
{  
  printf ("Hello, I am the first task\n");  
  fflush (stdout);
}

void user_hello_spg_2 (void)
{  
  printf ("Hello, I am the second task\n");  
  fflush (stdout);
}
```
To indicate the operating system we use, we add the property `Deployment::Execution_Platform` to the processor component. The declaration of the component `cpurm` should look like this:

```plaintext
processor cpurm
properties
  Deployment::Execution_Platform => Native;
end cpurm;
```

### 2.1.4 Test our implementation!

Now, you can test your system. Save your model and your C files. Then, click on your model in the file explorer and click choose the option `Generate code` in the `Ocarina` menu. The code would be generated. If not, a dialog box will report the errors.

Then, you can compile your code, using the menu `Build generated code` from the `Ocarina` menu, and run it using the `Run generated code` option.

### 2.1.5 Understand what is created

This example will help you to understand the semantics of AADL. In this example, there is one system. The system contains one process, which contains two threads.

As we said previously, the process is mapped into an UNIX process. So, when we generate the code, we create a UNIX process. In this process, we create two threads that execute the functions you just wrote in the C file. The threads are similar to the threads available in threading libraries like `linux pthreads`.

### 2.1.6 What could be expected now?

We saw that we can generate code and create concrete implementations from AADL models. However, the model we made is very simple and the generated code is not too difficult to write by hand.

Now, we will explore how to describe network communication with AADL and how we can automatically create code that manage network functionalities.

After that, we will explain why architectural models are so importance, giving some examples of properties that matter in the context of real-time embedded systems.

### 2.2 Network communications with AADL (exercise 2)

Now, we will make a simple example of two processes that exchange data through the network. The result is two process: the first sends periodically an integer to the other, which print the received number.

Now, we must introduce new AADL concepts: `features` and `connections`.

In AADL, `features` are added to components in order to show the interfacing facilities they provide. These `features` can be connected together. The connections of `features` models an interaction between component; the nature of the interaction depends on the kind of the components involved in the connection. For example, connections between two processes that are not located on the same computer will means that we
send data through socket. If the processes are located on the same computer, this communication could be achieved using shared memory. On the other hand, connection between a thread and a subprogram will model parameters on a subprogram call: it models what are the parameters of the subprogram and how the thread will make the call.

For some connections, we must provide additional properties. For example, when we model a connection between two processes, we must bound this property to a bus. It will indicates which bus the connection uses. It is required, because the bus can provide additional information on the connection and have its own requirements. And we must take them in account. However, we will not put a bus in the current model, we will do it in the next exercise.

2.2.1 What is missing in my model?

Nothing! But you have to write the application code by yourself. The issue here is that subprograms use parameters. Consequently, you have to take these parameters in account when you write your application code.

Look at the specification of the subprograms. Both have features that represent their parameters. The first subprogram has an `out` parameter. Consequently, in your function, this parameter will be a pointer because this kind of parameter allow us to change the content of the pointed value. The second subprogram has an `in` parameter. Consequently, you will simply write your function has its parameter is given by value.

2.2.2 What should I add?

Only write the application code for the subprograms. You have to write it in a file called `ping.c`. The mapping rules are the same than in the first exercise, the change between this exercise and the previous one is the communication between the component and the parameter in subprogram component.

The code should be look like this:

```c
#include <stdio.h>

int p=0;

void user_do_ping_spg (int *v)
{
    printf ("***SENDING_PING***%d\n", p);
    *v=p;
    p++;
    fflush (stdout);
}

void user_ping_spg (int i)
{
    printf ("***PING***%d\n", i);
    fflush (stdout);
}
```

You will note that you don’t know how you transport the data between the threads everything will be handled by the generated code.

```c
#include <stdio.h>

int p=0;

void user_do_ping_spg (int *v)
{
    printf ("***SENDING_PING***%d\n", p);
    *v=p;
    p++;
    fflush (stdout);
}

void user_ping_spg (int i)
{
    printf ("***PING***%d\n", i);
    fflush (stdout);
}
```
2.2.3 Test and run your system

Now, you can test your system. Save your model and your C files. Then, click on your model in the file explorer and click choose the option Generate code in the Ocarina menu. The code would be generated. If not, a dialog box will report the errors.

Then, you can compile your code, using the menu Build generated code from the Ocarina menu, and run it using the Run generated code option.

2.3 Understand the semantic of the process component (exercise 3)

Now, let’s take the model we made in the previous section and separate the threads on separate processes. We must declare a new process (let’s call it process B) that will contain one of the two threads. We must also add features on each process to exchange data between the two threads.

Moreover, we must bind the connection between the two processes to a bus component. Consequently, we must add a bus component to our model and bind the connection between the two processes to this bus.

The bus component provides information about the requirements of data transfer. When we declare a bus component, we must declare what kind of bus it is and how data transfer is achieved. This is done with the Deployment::Transport_API property.

To summarize, you must make the following changes on the previous model:

- Add a new process
- Move a thread to the new process
- Add/modify the features to model the transfer of data between the two threads.
- Add a bus component with the property Deployment::Transport_API set to BSD_Sockets.
- Bind the connection between the two processes with the Actual_Connection_Binding property.
- Add information in the runtime system (the processor component) to give location information (IP address and so on).

Finally, the modified model should look like this:

```
processor the_processor
features
  ETH : requires bus access Ethernet_Bus;
properties
  Deployment::location => "127.0.0.1";
  Deployment::Execution_Platform => Native;
end the_processor;

bus Ethernet_Bus
properties
  Deployment::Transport_API => BSD_Sockets;
end Ethernet_Bus;
```
— Processes —

process A
features
  Out_Port : out event data port Simple_Type;
end A;

process implementation A_Impl
subcomponents
  Pinger : thread P_Impl;
connections
  event data port Pinger.Data_Source -> Out_Port;
end A_Impl;

process B
features
  In_Port : in event data port Simple_Type;
end B;

process implementation B_Impl
subcomponents
  Ping_Me : thread Q_Impl;
connections
  event data port In_Port -> Ping_Me.Data_Sink;
end B_Impl;

— System —

system PING
end PING;

system implementation PING_Impl
subcomponents
  Node_A : process A_Impl;
  Node_B : process B_Impl { Deployment::port_number => 12002;};
  CPU : processor the_processor;
  the_bus : bus Ethernet_Bus;
connections
  bus access the_bus -> CPU.ETH;
  event data port Node_A.Out_Port -> Node_B.In_Port
    (Actual_Connection_Binding => (reference the_bus));
properties
  actual_processor_binding => reference CPU applies to Node_A;
  actual_processor_binding => reference CPU applies to Node_B;
end PING_Impl;

2.3.1 Test and run your system

Now, you can test your system. Save your model and your C files. Then, click on your
model in the file explorer and click choose the option Generate code in the Ocarina
menu. The code would be generated. If not, a dialog box will report the errors.

Then, you can compile your code, using the menu Build generated code from the
Ocarina menu, and run it using the Run generated code option.
2.3.2 Understand the changes

In this model, each threads is contained in a different process. Consequently, we will create two different process in the generated code. Moreover, we specify that the connection between processes is bounded to a bus, specifying its properties and requirements. In addition, we add information on runtime component to specify communication properties (location, port and so on).

Data transfer between the two processes is achieved with network sockets. The first process connects to the second using the TCP/IP protocol and the second processes receives data and transfer them to the thread. The code generator examine the model and look if the generated code needs to connect to remote nodes.

What we must notice here is that you don’t have to worry about network transfer and other complexities like that.
Chapter 3

Advanced use of AADL

3.1 Streaming music system

We showed in the previous exercise that we can define several processes and model data transfer between the processes. Now, we will illustrate our example and demonstrate that AADL models could be used to build real systems through a user-friendly example: a streaming music client/server.

We first describe what is a streaming server and client, giving its requirements and specificities. Then, we will model a such system and automatically generate the code.

3.1.1 What is a streaming server?

A streaming music server send small buffers that contains music to clients. It periodically sends data at a high rate. On the other side, the client receives the buffers and play it. A song is a very big file (about 50 to 100 Mbyte for an uncompressed music file), so, we have to split these songs to many little chunks and send it at a high rate.

3.1.2 Why this example illustrates real-time principles?

This example is very interesting, especially in the context of real-time systems. On the server side, timing requirements must be enforced: we have to play the music at a fixed rate and we don’t have to miss the deadline. It could be seen as a hard real-time system since a failure will impact clients. So, a failure or a misconfiguration will impact all clients.

For the client side, this is a good example between timing requirements enforcement will also impact the played music. If tasks are running too fast, the music will go too fast. If tasks are running too slow, music will not be heard like the original song.

This kind of example is especially relevant if we mix several time at the same time. The server process sends two different tracks of the same song at the same time while the client process receives it and play it. The timing requirements of all tasks will be important, because a failure in one task will delay one track and make the song less consistent. Consequently, the user will hear the impact of the failure.
We can also illustrate other requirements and the impact of each component on the system. Through this example, we will explain the impact of:

- Timing requirements of schedulable entities
- Latency and jitter issues
- Faults

All these topics belong to real-time embedded systems. Consequently, these example will depict their importance in a pleasant way (hear music).

### 3.2 Understanding delays and jitter (exercise 4)

Once we explained why our example fits with the requirements of real-time embedded systems, we can show our first model. The model (shown in figure 3.1) is taken from the flow latency model, defined in a technical report from the SEI [NEED CITATION /*FIXME*/]. We adapt this model for our goal: stream music between several processes and threads.

![Figure 3.1: The flow latency architecture, adapted for a streaming server](image)

#### 3.2.1 Understand our first streaming example

In this system, a thread sends data that contains the music. The data is then processed by three other threads that only copy the data and send it to the next process. Finally, the receiver process play the music contained in the data.

The main advantage of this system is that you can easily see the impact of communication between several nodes. Here, we have five nodes, each of them contains only one thread. Each node will send its data through the network. Consequently, for a music usage, a missed period or a latency will have impact on the music and the user will hear what is going wrong.
3.2.2 Understand the impact of Periods

3.2.3 Understand the impact of latency and jitter

3.2.4 Understand the difference between time-driven and event-driven tasks

3.3 Understanding faults and their impact

3.3.1 Understanding our second streaming example

Note: the model is provided in the archive of this tutorial.

We will define a simple architecture (depicted in figure 3.2 with two processes: one process (called sender process) sends the music buffers to the other process (called ). This second process will play the music at a fixed rate.

In the first process, we define two tasks, each task sends a separate track from the same song. The process sends these two tracks to the receiver process.

The receiver process has only one task that receives all music buffers and play them. The playing of the music buffers is achieved using dedicated subprograms.

We also model the data buffers with a dedicated data component. Each feature that sends or receives data uses this data classifier.

Each data chunk has a fixed size of 2048 bytes. A such buffer corresponds to 20ms of music. Consequently, each task must run at a rate of 20ms. We add these properties to the tasks.

As we depict in figure 3.3, each thread read a chunk in the music file that correspond to 20ms of music. Then, these threads send the data to the receiver process. This process receives the data, mix both music frequencies and play the file. Each thread is scheduled at 20ms. If this requirement is not enforced, the music will be slowly or will not be complete.

If we change the period of the thread and put the period of one sender thread to
read and sends
the first track
receive and play
all tracks
20ms
3 minutes
Period= 20ms
3 minutes
Both tracks
read and sends
the second track
Period= 20ms
20ms
3 minutes
Track one
Track two

Figure 3.3: Architecture of our streaming example

40ms, the music will not be complete and the synchronization between the two tracks will be lost. We depict this case in figure 3.4. Consequently, the timing requirements and their enforcement are crucial in safety-critical systems.

In addition, we can simulate the jitter we can meet when we use network interfaces. The jitter is the difference of the latency (i.e: the gap between higher and lower latencies). We can simulate it in our architecture and hear its impact on the implementation. We depict the impact of jitter in figure 3.5: music will not synchronized and needed information are not received on time.

### 3.3.2 Test and run your system

For this example, the model we provide is made with AADL version 2. You have to change your Ocarina preferences. Go in the preferences of the Ocarina plugin inside Eclipse to change that.

We provide the application code, as well as a file called userdefined.mk. This file defines variables dedicated to the compiler. These variables indicate which libraries are used and so on. This file is mandatory and you have to put it in the same directory than your model.

More, before testing your system, you must have two WAV files on your filesystem, in the same directory of your model. The file must have the name track1.wav and track2.wav.

Now, you can test your system. Save your model and your C files. Then, click on your model in the file explorer and click choose the option Generate code in the
Figure 3.4: Streaming architecture: impact of the period

Figure 3.5: Streaming architecture: impact of jitter
Ocarina menu. The code would be generated. If not, a dialog box will report the errors.

Then, you can compile your code, using the menu Build generated code from the Ocarina menu, and run it using the Run generated code option.

### 3.4 Hear impact of faults (exercise 5)

#### 3.4.1 Thread period

Now, we can change some properties of threads. Let’s change the period of the task that sends the second track. Change the value from 20ms to 40ms. The architecture will now be the same as in the figure 3.4. One sender thread send the data later than the other thread and the receiver cannot mix the tracks on time. Consequently, the music will be desynchronized.

To change your model, the declaration of the thread `P_Impl2` should be modified like this:

```plaintext
1 thread implementation P_Impl2
2 calls
3 seq:
4 | P_Spg : subprogram send_track2;
5 |}
6 : connections
7 parameter P_Spg.track -> track;
8 properties
9 | Period => 40 ms;
10 end P_Impl2;
```

Now, build your system again and run generated program. You will hear the impact of your changes: the second track is played at the same frequency as in the previous examples. However, the first tracks is too slow and the mix of the two tracks seems strange.

#### 3.4.2 Execution time of a thread, simulate jitter issues

Now, let’s change the execution time of a thread. We will add random additional execution time in the thread that sends the data. It will delay the sending of data.

It is a way to simulate the jitter for the connection between the sender and receiver threads. This additional execution time delays the sending of any data, which will be received later than we can expect. In real systems, the sending is performed on time by the sender side, but the jitter add additional time that delays data receiving. In both cases, a random time avoid the receiving of data on time.

To change your model, remove the changes introduced in the previous example and add the `Compute_Execution_Time` property to a sender thread in the first process. Set the value of this property to `10 ms .. 20 ms`.

With this declaration, we will add an additional time before sending any data. This additional time will change in each invocation of the task, its value has a value between 10ms and 20ms.

If you change the first task in the sender process, your model will look like this:
thread implementation P_Impl

calls
seq :
{
P_Spg : subprogram send_track1;
}
connections parameter P_Spg.track --> track;
properties
Compute_Execution_Time => 15 ms .. 20 ms;
end P_Impl;

Now, you can generate the code, rebuild generated programs and hear the impact of your changes: the first task take more time to be executed and data are not received on time. The song is desynchronized since both data are not received at the same time. What you hear does not seem to be the original song.

3.5 Hear impact of faults on your system

We saw that timing requirements have a significant impact on implementation and that many errors could be found at an architectural level. Now, we will see how faults could have an impact on the final implementation.

3.5.1 Add a random fault to your communication

Now, we want to add a random fault on communication port. For example, we will add a random fault on the task’s that send data buffers. To do that, add the SEI::MissRate

Figure 3.6: Streaming architecture: impact of faults
property to the ports of the task. This property indicates a percentage of fault on the port. We will specify that 50% of the communication may fail.

The architecture will look like figure 3.6. One thread will make a random fault and when fault happen, music is not sent. However, the music will still be read by the thread and synchronization will not be lost. We cannot predict when fault will happen since its is raised randomly.

To add a random fault in the component $P$, we must change the declaration of its port and add some property to indicate that that faults must be introduced. The declaration of the component should look like this:

```
thread $P$
defeatures
  track : out event data port Buffer {SEI::StreamMissRate => 0.5};
end properties
  Dispatch_Protocol => Periodic;
  Period => 20 Ms;
  Deadline => 20 ms;
  Cheddar_Properties::Fixed_Priority => 2;
end $P$
```

This declaration indicates that we introduce a random fault on the feature track. It means that half of the communication will be ignored.

Once you’ve modified your model, you can generate the code, build and run your system. Remember, you have to enable AADLv2 in Ocarina’s preferences, add the file userdefined.mk and two WAV files called track1.wav and track2.wav. You will hear that many parts of the song are not played, because of the introduced faults. However, when the receiver is playing the both tracks, they are synchronized, whereas in other cases, synchronization was lost.

Now, you can play with this property and see that the lower the value is, better the song is. We see the impact of faults on music, seeing its frequencies and spectrum. The overall spectrum (depicted in figures 3.7 and 3.8) does not really changes, in the sense that it’s difficult to see impact of faults. However, if we consider a small time slice, we may see the impact of these faults. We show the spectrum of the first 70ms of the song in figures 3.9 (no fault) and 3.10. We can see that music spectrum differs when faults are introduced. In fact, sometimes, when a fault is raised, the same track is played. Consequently, the receiver thread mixes two tracks, but one track is the same than in the previous mix, which change the music played.

![Figure 3.7: Overall music spectrum, no fault](image)

### 3.6 Conclusion

Through these examples, we depict the importance of architectural models. We demonstrate that architectural properties have a direct impact on implementations. We em-
We also saw that a code generator could help system designers as well as developers: the code is automatically generated, the specifications enforced, and bugs introduced by hand-written code are avoided. In a Model Driven Environment (MDE), we must automatically generate the code from the specifications. It avoids any interpretation and ensure that requirements are enforced.

These examples demonstrate that architectural models help system designers. We demonstrate the importance of architectural models but we must detect them early in the development process. To do that, we can automatically check the architectural model. Many plug-ins are available for OSATE\(^1\) in order check timing requirements, latency, jitter and other crucial properties of your system.

\(^1\)see http://www.aadl.info
Bibliography