AADL2OSGi - a short introduction

1. Goal

> Make it easy for developers to transform their AADL models into a running prototype that conforms to the semantics of their model (as far as this is possible with Java/RTSJ/OSGi).
>
> — aadl2osgi

2. Terms and Basics

2.1. AADL

Great modeling language to describe performance-critical systems. Further information can be found at

- http://www.aadl.info/aadl/currentsite/
- https://wiki.sei.cmu.edu/aadl/index.php/Main_Page
- https://github.com/osate (Reference Tooling)

2.2. Xtend

Xtend[1] is a flexible and expressive dialect of Java, which compiles into readable Java 5/6/7/8 compatible source code. You can use any existing Java library seamlessly. The compiled output is readable and pretty-printed, and tends to run as fast as the equivalent handwritten Java code.

Don't be shy… if you can write Java code, then you can write Xtend code as well ;)

We use Xtend's Active Annotations[2] to turn simple Xtend classes and fields into code that conforms to the semantics of AADL.

[1] http://www.eclipse.org/xtend/
[2] https://eclipse.org/xtend/documentation/204_activeannotations.html

2.3. OSGi

> The **OSGi specification** describes a modular system and a service[3] platform for the Java[4] programming language that implements a complete and dynamic component model[5], something that does not exist in standalone Java/ https://en.wikipedia.org/wiki/Virtual_machine[VM] environments. Applications[6] or components, coming in the form of bundles[7] for deployment[8], can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot[9]; management of Java packages[10]/https://en.wikipedia.org/wiki/Class_(computer_science)[classes] is specified in great detail. Application life cycle management is implemented via APIs that allow for remote downloading[11] of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.
>
> — Wikipedia

OSGi is the platform in the background to which we compile our annotated classes and fields. The parts that are important for you as a user will be highlighted later in this document, but if you are curious I recommend the following sites to get a first glimpse of what is possible with OSGi:

- http://enroute.osgi.org/
- http://blog.vogella.com/2016/02/09/osgi-bundles-fragments-dependencies/
- http://blog.vogella.com/2016/06/21/getting-started-with-osgi-declarative-services/
- http://blog.vogella.com/2016/09/26/configuring-osgi-declarative-services/

[3] https://en.wikipedia.org/wiki/Service_(systems_architecture)
[4] https://en.wikipedia.org/wiki/Java_(programming_language)
[5] https://en.wikipedia.org/wiki/Component_model
[6] https://en.wikipedia.org/wiki/Application_software
[7] https://en.wikipedia.org/wiki/OSGi#Bundles
[8] https://en.wikipedia.org/wiki/Software_deployment
[9] https://en.wikipedia.org/wiki/Reboot_(computer)
[10] https://en.wikipedia.org/wiki/Java_package
[11] https://en.wikipedia.org/wiki/Downloading

- http://blog.vogella.com/2017/02/13/control-osgi-ds-component-instances/
- http://blog.vogella.com/2017/02/24/control-osgi-ds-component-instances-via-configuration-admin/

3. Installation

I chose to publish this setup as a VirtualBox image as it is a little bit tricky to get everything up and running with the JamaicaVM (our targeted RTSJ implementation). Therefore:

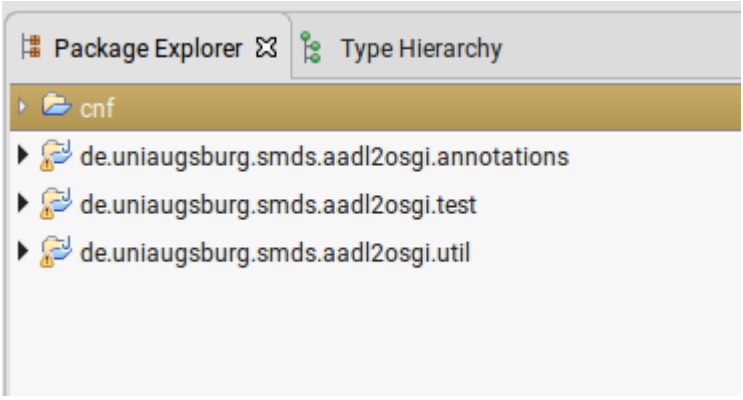
- Download VirtualBox[12] and install it on your machine
- Download VB image[13] and import it to VirtualBox (just doublecklick on it, then follow the wizard)

After you have imported the VB image you can start up the virtual machine

This image is a 64-bit Linux. Your PC must support hardware virtualization in order to be able to run this image! On many PCs hardware virtualization is turned off within BIOS → Turn it on ;)

4. Project Structure

On the desktop of the virtual machine you should see an Eclipse installation, whose package explorer should look something like this after startup:



12 https://www.virtualbox.org/
13 https://drive.google.com/file/d/0Bw1daHW2hcyZbDJ4NDZaSEx3YkE/view?usp=sharing

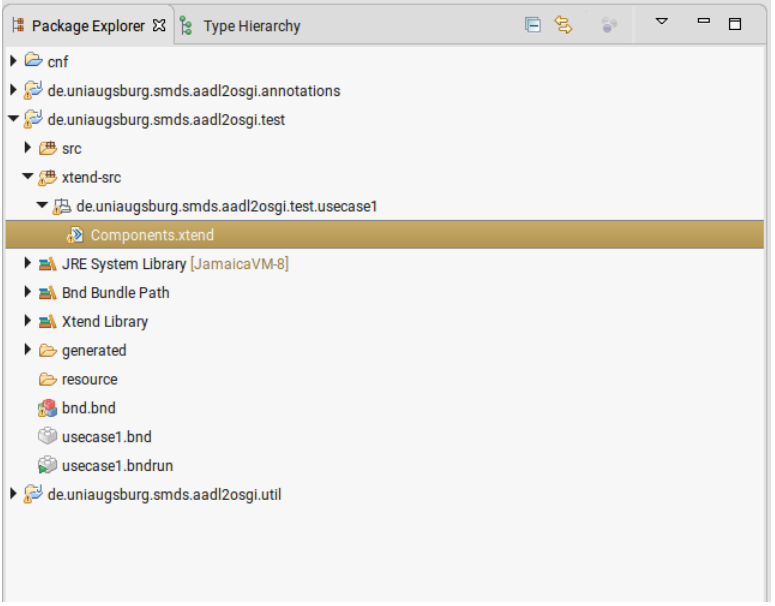
4.1. de.uniaugsburg.smds.aadl2osgi.annotations

This project contains the Active Annotations which enable us to transform plain Xtend classes and fields into a running OSGi application. All annotations are named like their corresponding AADL component counterpart, e.g. Thread = @AADLThread.

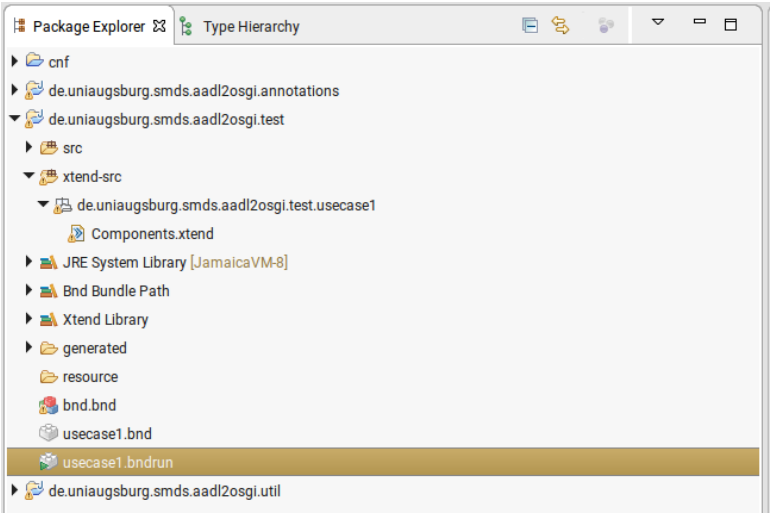
Each annotation is kept in its own Xtend file, which contains the annotation and its corrresponding processor. The util package contains some classes that help with navigation between annotations, creating OSGi references in the generated code and so on.

4.2. de.uniaugsburg.smds.aadl2osgi.test

This project contains a sample application which is written down in the Comopnents.xtend file:



It also contains the corresponding bndrun file:



4.3. de.uniaugsburg.smds.aadl2osgi.util

This project only contains shared interfaces that are needed during generation and during runtime alike. Those interfaces are split into two packages curretnly: fw and user. fw contains the interfaces which can be seen from the classes of our generated framework and user contains the interfaces that are meant for the user to use when he accesses classes generated by the active annotations, e.g. while accessing an InDataPort.

5. AADL in Xtend/Java - Easy as pie

The aforementioned sample application consists of a Process (MyProcess), which contains two Threads (MySender, MyReceiver). MySender has an out data port, MyReceiver an in data port and both are connected via a port connection. Have a look at the next figure to see how easy it is to declare such a system in Xtend.

Components.xtend

```
package de.uniaugsburg.smds.aadl2osgi.test.usecase1

import de.uniaugsburg.smds.aadl2osgi.annotation.component.process.AADLProcess

@AADLProcess
class MyProcess{

    @ThreadSubcomponent
    MySender sender

    @ThreadSubcomponent
    MyReceiver receiver

    @PortConnection(
        dataType = String,
        source = 'sender.outport',
        target = 'receiver.inport'
    )
    Object myCon1
}

@AADLThread(
    parent = MyProcess,
    periodMs = 2000
)
class MySender{

    var counter = 0

    @OutDataPort(
        dataType = String
    )
    Object outport

    @InitializeEntrypoint
    def init(){
        start
    }

    @ComputeEntrypoint
    def test(){
        val data = 'some data'
        outport.data = data
        println('''Sending «data»''')
    }

    @FinalizeEntrypoint
    def deinit(){
        stop
    }
}

@AADLThread(
    parent = MyProcess,
    periodMs = 2000
)
class MyReceiver{
    var counter = 0

    @InDataPort(
        dataType = String
    )
    Object inport

    @ComputeEntrypoint
    def test(){
        println('''Receiving «inport.data»''')
    }

    @InitializeEntrypoint
    def init(){
        start
    }


    @FinalizeEntrypoint
    def deinit(){
        stop
    }
}
```

Right now these annotations are the only ones that are supported (it's a prototype ;)), but we're working on the next iteration. Stuff that is on our roadmap are for example: Data/Subprogram components, data/subprogram access connections and modes.

5.1. The example in detail

@AADLProcess

This one does not really do much. In the background this class is translated into an OSGi declarative service with references to its threads, which will also be translated into declarative services. The declared port connection is translated into a separate class. There are also no parameters that can be set on @AADLProcess, so that's it.

@AADLThread

This annotation is the most complex one as threads usually contain the running code, have to define a dispatch protocol, a period, a deadline and stuff like that. Therefore, the parameters that can be changed are:

- periodMS: the millisecond part of the period of this thread (default = 1000)
- periodNS: the nanosecond part of the period of this thread (default = 0)
- deadlineMs/deadlineNs: see period (defaults: 500 / 0)
- priority: the priority of this thread (default 11 (that's the lowest hard real-time priority in RTSJ))
- dispatchProtocol: can be periodic, sporadic, etc. but currently we only support periodic threads (default = periodic)

An addtitional parameter on this annotation is parent. This one mustbe set to the class that declares this Thread to be a subcomopnent of it. This parameter has nothing to do with the AADL model, but is needed by our translation.

This class is translated into several more classes, e.g. a framework-class, phaser-class, user-class. The details of how and why will be explained in a paper to come.

Within a class annotated with @AADLThread one should annotate at least one method with **@ComputeEntrypoint** as this is the code that is

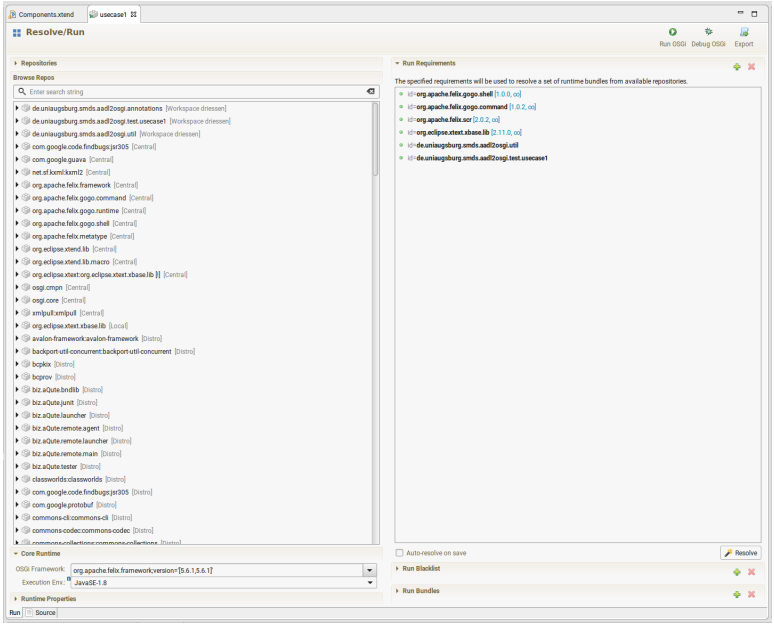
executed within each dispatch cycle. The annotations **@InitializeEntrypoint** and **FinalizeEntrypoint** can be used likewise. They are executed on startup and shutdown of this thread.

@InDataPort / @OutDataPort / @PortConnection

Those annotation do what their name implies, they create data ports an port connections. InDataPorts can only be read, OutDataPorts can only be written. PortConnections will never be used by the user directly, as the framework takes care of transmitting data over the defined connections. For each you can define the datatype that they hold (currently only Integer, Double, Float, String, Boolean, Long) and should be notified if there is a mismatch between the datatypes of outgoing/ingoing ports or the connection over which the data should be transmitted. You also can define at which point in time data shall be sent or received via the annotation parameters inputTime, outputTime.

5.2. What to try

First start up the whole thing by going to usecase1.bndrun and clicking on **Run OSGi**. Now you should see the console output generated by the two @ComputeEntrypoint methods. One is sending and the other one receiving.



In your console you can type **lb** (for list bundles), hit enter and should get an output similar to the follwoing one:

```
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (5.6.1)|5.6.1
    1|Active     |    1|Guava: Google Core Libraries for Java (18.0.0)|18.0.0
    2|Active     |    1|de.uniaugsburg.smds.aadl2osgi.util (0.0.0)|0.0.0
    3|Active     |    1|Apache Felix Gogo Command (1.0.2)|1.0.2
    4|Active     |    1|Apache Felix Gogo Runtime (1.0.0)|1.0.0
    5|Active     |    1|Apache Felix Gogo Shell (1.0.0)|1.0.0
    6|Active     |    1|Apache Felix Log Service (1.0.1)|1.0.1
    7|Active     |    1|Apache Felix Metatype Service (1.1.2)|1.1.2
    8|Active     |    1|Apache Felix Declarative Services (2.0.2)|2.0.2
    9|Active     |    1|Xbase Runtime Library (2.11.0.v20170124-1424)|2.11.0.v20170124-1424
   10|Active     |    1|de.uniaugsburg.smds.aadl2osgi.test.usecase1 (0.0.0)|0.0.0
```

Now you could try to type something like **stop 10** (10 is the number of the usecase1 bundle) which should stop this bundle and therefore stop the output of both, sender and receiver. You also could type **start 10** to restart this bundle.

One great thing about OSGi is the possibility to rewrite your code and redeploy it into the running system. You can now go to Components.xtend and write something like **println('Activated Sender')** into the @InitializeEntrypoint method of your sender class. (You can also do this likewise for the receiver and both ther @ FinalizeEntrypoint methods to get a better picture of their lifecycle). Now save everything and in your console should appear the text you just typed in your output statement.

Each time you now start and stop the bundle the text of both methods should be printed to the console.

You can also go a little step further and change the periodMs of one or both threads, save everything and will see that (at runtime) your code gets reloaded and will reflect the changes you just made to the thread.