ACS Characteristic Component Generator

Software Operations

Exported on 03/01/2019

Table of Contents

1 Introduction

This document contains detailed information about the ACS Characteristic Component generator developed during the 2019 summer internship in the months of January and February. In the introduction there is a description of the basic usage of the generator and a short description of the generated directory structure and files. In the definitions section the meta-model used by the generator is described, and the way that the BACI types and properties are modeled is described in more detail. In the definitions section there is also the specification for the DevIO so that it might be used with the generated code. In the detailed system description there is more information about how the generator was developed. In case of wanting to make changes to the model or generator this section should be useful. The final section describes future work that could improve the system. At the end there are the references and a glossary.

This project aims to facilitate the implementation and deployment of ACS characteristic components by generating all the configuration files and most of the necessary implementation files. The generation is based on an instance of a meta-model that should be created following a Characteristic Component definition that consists of its Actions, Attributes, Properties and Instances

The system consists of 4 main eclipse projects, in 3 of these projects the code is mostly generated by EMF with some handwritten changes to add functionality, while the other one is the Acceleo generator, for which the templates are handwritten but also has a Java file that is generated. More detailed information about each of the eclipse projects can be found in the System description section. From the user's perspective the system has 2 parts, the editor and the generator. With the editor an XMI file can be created and changed and the instance can be validated. This file stores the necessary information to generate the required files according to the component's definition.

1.1 Basic Usage

By the time of the creation of this document the editor and the generator are only available as eclipse projects. In the future they could be launched as an Eclipse Plug-in and a runnable jar file respectively. Despite this, the system can be used to its full functionality as it is, by importing the projects and setting the launch configuration of the editor and the generator. The launch configuration for the Acceleo project is provided in the project's folder but could be changed to use a different XMI file if desired.

To use the generator in its current state the following steps are required:

- Install Eclipse 4.6.
- Install EMF.
- Install Acceleo.
- Import the model, edit and editor projects.
- Configure the editor to be launched as an Eclipse plug-in.
- Launch the editor as an Eclipse plug-in.
- In the runtime instance of Eclipse, import the Acceleo project.
- In the runtime instance, import the provided example project, or create a new project (*).
- Configure the Acceleo project to be launched as an Acceleo Application(**).
- Edit the instance file with the editor (***).
- Optionally validate the instance (Right click on Characteristic Component in the editor → Validate).
- Run the Acceleo Application.

(*) To create a new project for a model instance: Create a new general project (File → New → Other... → General > Project). Create a folder inside the project (File → New → Folder). Finally create a BaciCodeGen Model (File → New → Example EMF Model Creation Wizards > BaciCodeGen Model). As model Object, select 'Characteristic Component'.

(**) You might want to use the provided configuration file (org.eclipse.acceleo.acsBaciCodeGen.generator > runConfig > Acceleo Code Generation.launch). Or edit it, or create a new one (Run → Run Configurations...). The fields that you might want to edit are the model file (the .bacicodegen) or the target folder where the generated files are stored.

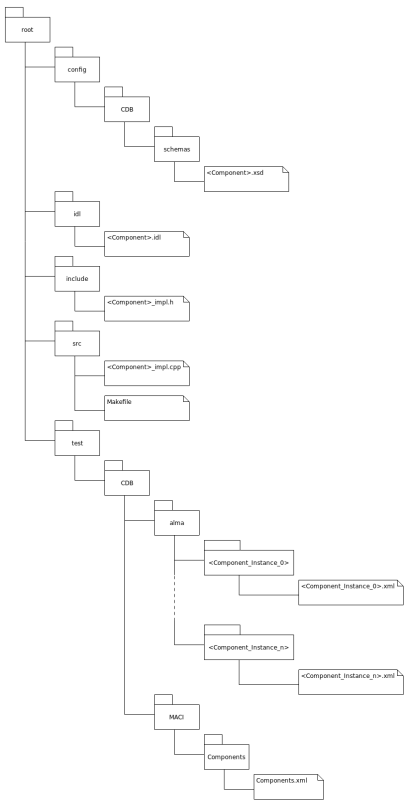
(***) Open the .bacicodegen file with BaciCodeGen Model Editor (default). You should probably take a look at the definitions and editor sections in this document first. The component instances should be the last object to be created, since they are generated from the rest of the component's definition.

With a correct definition of the component and a proper creation of the instance the generator will create the file structure for the component, and the configuration for the component and its instances completely. The generation does not consider the initialization of DevIO variables which are considered arbitrary, nor the implementation of actions and lifecycle methods since this could be virtually anything. This implementation is the user's responsibility and it can be achieved by editing the .cpp file in the defined user code blocks. These blocks are marked by comments "//Start of user code <BlockID>" and "//End of user code" and will not be overwritten if the generator is run again.

After this steps are completed the user has to initialize the DevIO variables and implement any desired functions in the C++ implementation file. Although the implementation of functions is optional, the initialization of the DevIO variables is necessary to make the component functional.

1.2 Generated Files

The generated files follow the ALMA directory structures as is specified in the BACI Device Server Programming Tutorial. For a <Component> with component instances <Component_Instance_0> to <Component_Instance_n>, the generated files and directory structure are the following:



<Component>.xsd: XML schema for the component. Properties and Attributes are declared here. Imports BACI and CDB schemas.

<Component>.idl: IDL definition for the component (as COB). Properties and Actions are declared here. Also declares the module and interface name, and may add a prefix. Includes BACI IDL.

<Component>_impl.h: Header for the implementation file. Declares Lifecycle methods, Properties, Actions, SmartPropertyPointers, and DevIO variables.

<Component>_impl.cpp: C++ implementation of the component. The implementation for all the functions declared in the header can be done here. DevIO variables must be initialized to make the component functional. User implementation can be done in user code blocks marked by comments. Any changes outside this blocks will be overwritten if the component is generated again.

Makefile: Makefile for component installation.

<Component_instance_i>.xml: Initial configuration for a component's instance. Has the values of Attributes and Characteristcs for the instance. There is one folder with one xml file for each instance. When changing names or removing instances the folders of old instances should be manually removed.

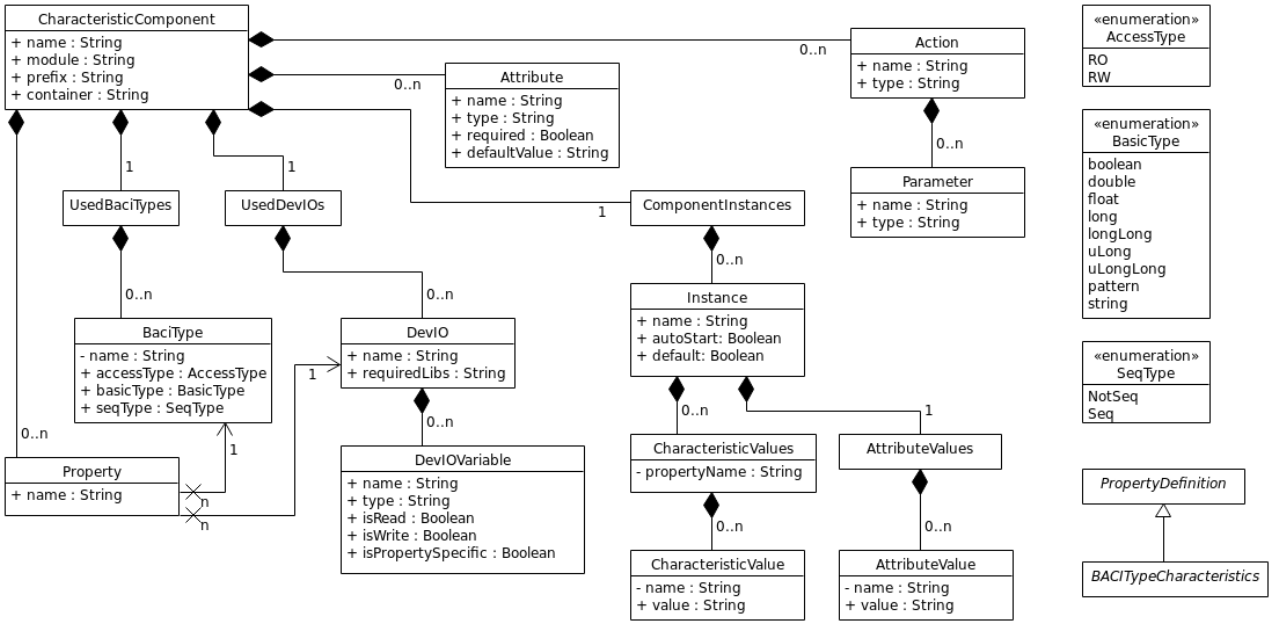
Components.xml: COB configuration for the component. Declares each instance of the component with their container, implementation language and the attributes Autostart and Default.

2 Definitions

2.1 Meta-model

The following is the meta-model created for the generation. From this model an instance can be created using the editor, and from this instance the code is generated. This model was created with the intention of making the user's interaction and template creation simpler and so, has many things that are just for this purpose. All the attributes can be accessed in the implementation, the ones labeled as private are derived from other objects in their class creation and cannot be modified by the user in the editor.

The main class is the characteristic component, which contains all other classes. It directly contains the classes Attribute, Property and Action, which are the main components of a characteristic component. It also holds containers for the BACI types and DevIOs used by the properties, and the instances that hold the configuration values for the deployment of the component.



2.1.1 Class description

CharacteristicComponent

Contained by: -.

Contains: Action, Attribute, ComponentInstance, Property, UsedDevIO and UsedBaciType.

Attributes: name, module, prefix and container.

Main class of the model. Represents a characteristic component, and so, its attributes are the ones that define a component and it's composed of all the elements that a characteristic component has.

Action

Contained by: CharacteristicComponent.

Contains: Parameter.

Attributes: name, type.

Represents an action of the characteristic component. Its type corresponds to the return type of the C++ function that will be implemented, and the contained class represents the function's parameters.

Parameter

Contained by: Action.

Contains: -.

Attributes: name, type.

Represents a parameter of an action which will be implemented as a C++ function, so it has a name and a type.

Attribute

Contained by: CharacteristicComponent.

Contains: -.

Attributes: name, type, defaultValue.

Represents an attribute of the characteristic component. It is declared in the components XML schema and it's default value is used when creating an Instance in the editor. It is part of the CDB so its value can be obtained from it in the implementation.

Property

Contained by: CharacteristicComponent.

Contains: -.

Attributes: name.

Represents a property of the characteristic component. It has a reference to a BaciType which determines its return type, the type of smart pointer that it will use and the DevIO objects that will be instantiated for it.

UsedDevIOs

Contained by: CharacteristicComponent.

Contains: DevIO.

Attributes: -.

Container for the DevIOs that are used by the component. Many properties might use the same DevIO, so its better to have them separately, and this way it's also easier to iterate through the DevIOs in the templates.

DevIO

Contained by: UsedDevIOs.

Contains: DevIOVariable.

Attributes: name, requiredLibraries.

Represents a DevIO used by the component. Its name must comply to the specification in this document to be usable. Its required libraries are added to the makefile as one string.

DevIOVariable

Contained by: DevIO.

Contains: -.

Attributes: name, type, isRead, is Write, isPropertySpecific.

Represents a variable that will be used as an argument for a DevIO. Multiple variables may be generated depending on its attributes isRead, isWrite and isPropertySpecific. All DevIOVariables must be at least read or write, otherwise they will not be generated. Property specific variables will generate a different variable for each property for which the DevIO is used. Variables that are not property specific, are common to all the devIO objects created, they may be common to the read objects, to the write objects or to both. Property specific variables will generate one variable for each property and for each access type even if they are read and write, while common variables will generate only one variable that will be used for read, write or both types of objects. For more detailed information see DevIO specification.

UsedBaciTypes

Contained by: CharacteristicComponent.

Contains: BaciType.

Attributes: -.

Container for the BaciTypes that are used by the properties. Many properties might use the same BaciType, so its better to have them separately, and this way it's also easier to iterate through the BaciTypes in the templates.

BaciType

Contained by: UsedBaciTypes.

Contains: -.

Attributes: name, accessType, basicType, seqType.

BaciType to be used by the CharacteristicComponents properties. Its name is derived from its other attributes so it can't be edited by the user. The other attributes are enumerations of the subtypes that determine a BACI type. For more detailed information see BACI types.

ComponentInstances

Contained by: CharacteristicComponent.

Contains: Instance.

Attributes: -.

Container for the instances of the Characteristic Component.

Instance

Contained by: ComponentInstances.

Contains: AttributeValues, CharacteristicValues.

Attributes: name, autoStart, default.

Represents an instance of the CharacteristicComponent. An instance has values for the characteristics, and attributes of the component. Its attributes autoStart and default determine whether the instance has the attributes AutoStart and Default in its configuration.

AttributeValues

Contained by: Instance.

Contains: AttributeValue.

Attributes: -.

Container for the attribute values of a characteristic component's instance.

AttributeValue

Contained by: AttributeValues.

Contains: -.

Attributes: name, value.

Represents an attribute of an instance of a characteristc component for which a value is given. Its name is not editable by the user because it corresponds to an already defined Attribute, and its value is initially the default value specified in the Attribute with the same name.

CharacteristicValues

Contained by: Instance.

Contains: CharacteristicValue.

Attributes: propertyName.

Container for the characteristic values of a characteristic component's instance. Its propertyName is a derived attribute from a corresponding property, it's not editable by the user.

CharacteristicValue

Contained by: CharacteristicValues.

Contains: -.

Attributes: name, value.

Represents an attribute of an instance of a characteristc component for which a value is given. Its name is not editable by the user because it corresponds to an already defined Attribute, and its value is initially the default value specified in the Attribute with the same name.

PropertyDefinition

Contained by: -.

Contains: -.

Attributes: -.

Abstract class that represents the definition of a property as specified by BACI. This class is used by the instance to create objects of different kinds of property definitions that correspond to the BACI type of each property.

2.2 BACI Types

As defined in the ACS Supported BACI Types document, these can be RO(Read Only) or RW(Read Write); sequences; and have different basic types, such as boolean, double, string, etc. Because of this, in the generator model they are represented as having 3 sub-types: an access Type (RO or RW), a basic Type (boolean, double, float, long, longLong, uLong, uLongLong, pattern or string) and a sequence type (sequence or not sequence). The sub-types are modeled as enumerations so that only valid values are used. Despite this, from the 36 BACI types that could be specified by the combination of the sub-types, there are several that are not yet implemented and might be never implemented. Because of this an error is thrown by the editor when creating an instance of a component that uses an invalid BACI type. For more information about how this is done, see Handwritten code in the System description section.

2.3 BACI Properties

The definition of BACI properties is obtained directly from the XML Schema in which the definitions are implemented. Te way to include this in the meta-model is to create a Ecore model from the .xsd file, and then load this model as resource to the main model. After loading it as a resource, the hole package can be copied and pasted into the main package. This allowed to make the creation of the property characteristics model much faster, and exactly like it is defined, since the definition's inheritance and restriction of the characteristics make it quite intricate.

In the generated model, the property definitions are created as classes and the characteristics are attributes of the classes. This attributes are of the java type that corresponds to the characteristic's xs type (xs:string → String) and have the characteristics default value as default value literal. Since EMF models have methods to get the meta-class features, this model can be used to generate the configuration of a component's instance automatically from the definition of the component.

2.4 DevIO specification

In order to have the generation of the component independent of the DevIO, some specifications were defined for the names of the DevIO classes. To generate a component that uses a DevIO, the DevIO should be created with this specifications in mind. It is also important to understand the definition of the DevIO variables to make the implementation simpler. All of this is necessary because the implementation creates objects of the DevIO classes and the generator model considers that the arguments of its constructors are somewhat arbitrary.

For a DevIO in the model with the name <SomeDevIO> the following will be assumed:

- The DevIO's header is be named "<SomeDevIO>_devio.h".
- The DevIO consists of a namespace "<SomeDevIO>".
- This namespace has classes "<SomeDevIO>_read" and "<SomeDevIO>_readWrite" (Only required when used).

Besides this, the constructors of the classes must have receive the parameters defined as DevIOVariables and inherit from the correct DevIO Type. This DevIO type is related to the *basicType* of the property that will be using it, they should be the same.

2.4.1 DevIO Variables

Two main kinds of DevIO variables can be created: common variables and property specific variables. They represent the variables that will be used by the DevIO but are also related to the kind of argument that is expected. Although both of these can be read or write, this attributes have different meanings for one or the other. A common variable represents an argument that will be *common* to read or write objects for each property, while a property specific variable represents an argument that will be *specific* to read or write objects and for each property. Common variables are always the first.

Example:

A DevIO named "mqtt" that is used by ROdouble and RWdouble properties must have:

- A header: "mqtt_devio.h".
- A namespace: "mqtt".
- Classes: "mqtt_read" and "mqtt_readWrite"

If the DevIO uses 4 variables for 4 properties:

- var1: common, read
- var2: common, read, write
- var3: property specifc, write
- var4: property specific, read, write

- prop1 and prop2: ROdouble
- prop3 and prop4: RWdouble

The generated code for the DevIO object creation in the implementation C++ file will be:

prop1_devio_m = new mqtt::mqtt_read(var1, var2, r_prop1_var4);

prop2_devio_m = new mqtt::mqtt_read(var1, var2, r_prop2_var4);

prop3_devio_w = new mqtt::mqtt_readWrite(var2, w_prop3_var3, w_prop3_var4);

prop4_devio_w = new mqtt::mqtt_readWrite(var2, w_prop4_var3, w_prop4_var4);

3 Detailed system description

3.1 ECore model

The ecore model is the implementation of the meta-model's design. This implementation is done through the definition of classes, attributes and references. All the relations in the meta-model are represented as references, and there are some references that are in the ecore model but are not present in the class diagram. The references from Instance to its container and ComponentInstances to CharacteristicComponent for example, are used to create objects within the Instance to let the user define Attribute and Characteristic values. There are also aspects of the model that are not specified here, like the read-only attributes. These are given the read-only property in the Genmodel file.

Through this file, some model restrictions can be specified, such as required objects and attributes. This restrictions can be checked when using the generated editor and should help define a working component. The generator however does not check this restrictions so it's important that the user takes care of this while editing the meta-model instance.

The ecore model is stored in an XMI file that can be edited with a dedicated EMF editor. This allows the meta-model creation to be much faster and simpler but it also might create some problems, for example, when editing an already created file, some attributes might not be changed as expected. The XMI file can be edited in any text editor as well.

The user does not need to modify the ecore model but it's necessary that it is properly understood to generate the desired component correctly.

3.2 Genmodel

The genmodel file is created using a creation wizard that allows the selection of an ecore model. This file allows the creation of an instance of the meta-model (a model) by generating the required packages for an eclipse plugin. The generated plugin includes a creation wizard for model instance files and a graphic editor to modify them. The created instances are stored as XMI files that can also be edited in a text editor though this is not recommended for several reasons, the main ones being the automatic generation of instance objects (for more details read the Editor section) and the fact that the user would need to know the model almost perfectly.

The generation of the packages can be done by opening the Genmodel file with the EMF editor, right clicking on the model and selecting Generate All. After modifying the meta-model the packages can be generated again. If generated files were modified, the changes can be kept by changing the "@generated" tag in the javadoc to "@generated NOT".

Sometimes changes in the meta-model are not reflected properly in the genmodel file. If this happens the XMI file can be reviewed in a text editor or deleted and created again from the ecore model. Note that the later means losing any changes made to the genmodel specifically.

3.3 Generated Java code

3.3.1 Packages

For each package in the Ecore model, three java packages are created in the model project, and 3 other projects are created with several java packages each. The packages generated for the model are the meta-classes definition, the

meta-classes implementation, and a utility package. The other projects are the edit, editor and test. In this case, only the model packages were modified and only the editor package is used as an eclipse plugin to edit model instances. The edit package is used by the editor, but the test package is not necessary at all.

3.3.2 Handwritten code

Most of the modifications to the generated meta-classes was made to the getters of the attributes or references in the implementation of the corresponding classes or containers. This methods are called several times, but the initialization of the objects is done only when the references have not been set. This is checked using the generated "isSet<ReferenceName>" methods, which are created when the references have the property "Unsettable" (meaning that they can be unset). The following files in the src folder of the project vvaras.acsBaciCodeGen.model were edited to implement the creation of derived variables and objects for the instances:

- baciCodeGen.impl/**BaciTypeImpl.java**
- baciCodeGen.impl/**InstanceImpl.java**
- baciCodeGen/**AttributeValues.java**
- baciCodeGen.impl/**AttributeValuesImpl.java**
- baciCodeGen/**CharacteristicValues.java**
- baciCodeGen.impl/**CharacteristicValuesImpl.java**

The details of modifications made to each file are the following:

BaciTypeImpl

The method getName was modified to have the BaciType's name attribute be derived from the sub-types. This is done by obtaining the literals of the enumerations accessType, basicType, and the vale of seqType. The strings from the accessType and basicType are concatenated, and if the value of the seqType is one, the string "Seq" is added. The resulting string is returned.

InstanceImpl

The method getAttributeValuesContainer() was modified to initialize the container for the AttributeValues. If the container has not been set, a new container is created with the AttributeValuesImpl() constructor, then the reverse reference for the container is set and finally the containers method for constructing the attributes is called. As default, the method returns the attributeValuesContainer.

In the same way, the getCharacteristicValuesContainer() method was modified. The difference in this case is that because the characteristicValuesContainer reference of the instance can have multiple values, the reference is implemented with an EList of CharacteristicValues, instead of a single object. The creation of these CharacteristicValues is related to the properties of the component. A CharacteristicValues object is created for each property, and the Characteristics that it will contain depend on the BaciType of the property. With the reverse containment references, the property EList is obtained, a characteristicValues container is created for each, and a container's method is called to initialize the characteristics. Each object is added to the EList and as by default, the EList is returned.

AttributeValues

The method setInstanceAttributes() was declared.

AttributeValuesImpl

The method setInstanceAttributes() was implemented. If the reference is null, it is initialized. Then the list of the component's attributes is obtained through the reverse containment references. An AttributeValue object is created for each attribute using the AttributeValueImpl() constructor. The name and the default value of each AttributeValue are set from the attribute. The AttributeValues are added to the EList and the Elist is returned.

CharacteristicValues

The methods setInstanceCharacteristics() and getBaciTypePropertyDefinition() were declared.

CharacteristicValuesImpl

The method setInstanceCharacteristics() was implemented. This method receives a Property as an argument. If the reference is null, it is initialized. A PropertyDefinition object is created using the method getBaciTypePropertyDefinition(). To call this method it is necessary to first obtain the property's BaciType. Depending on the BaciType, a specific kind of property definition will be created. From the property definition, all the EAttributes are obtained. EAttributes are the meta-model attributes of the object's class. This attributes represent the property's characteristics according to the BACI definition. For each of the attributes a CharacteristicValue is created, its name and default value set, and its added to the EList. The EList is returned.

The method getBaciTypePropertyDefinition() was implemented. It receives as arguments three integers that correspond to the accessType, basicType and seqType of a BaciType. From this arguments, using switches, a PropertyDefinition object is created and returned. There are several cases for which the BaciType is not implemented. In this cases an unsupported operation exception is thrown. The creation of the propertyDefinition is done calling a method of a BACIPropertiesFactory because it's a class defined in another package.for

3.4 Editor

The editor is a generated Eclipse plugin that allow the user to create instances of the meta-model for the generation with Acceleo. When the plugin is being used, a creation wizard for the instance files is available. After creating an instance file, it can be modified with the editor.

To create an instance file, first create an empty project with a folder in it. Then go to: File → New → Other → Example EMF Model Creation Wizard > BaciCodeGen Model. Click next and select the parent folder and the file name. The file is an XMI file and its extension has to be .bacicodegen. If creating a full instance of the meta-model, select CharacateristicComponent as the Model Object and click the Finish button.

Other objects can be created to be used as resources in other instances. For example, a DevIO could be defined as a separate file and used in various instances for properties with the same BaciType. To do it right click in the editor window, select Load Resource... and find the desired resource. It will appear as a different file in the same editor window and the objects in it can be referenced by objects in the instance. The user should be careful when making changes to loaded resources, if a resource has to be modified for a particular instance it might be better to copy it into the model.

With the editor the instance can be validated. This is done right clicking on the instance resource and selecting Validate. The validation checks that the required attributes are set and the necessary objects are defined. For example, to generate a component it is mandatory to have at least one instance created. If there are no Instances for the component the validation the validation will report that the feature has 0 values when it should at least have 1. Its important that the model is validated to do a useful generation. The restrictions for the validation are stronger than the ones for the generation. A model might be generated without errors while having validation problems but it will most likely not even get compiled. Although a successfully validated instance might not always generate a functional component, it is a way to ensure that it fulfills minimum requirements and might even be useful to guide the instance creation.

Important: The instances should be created when the rest of the component is complete. If changes to the attributes or properties are made after creating the instances , the instances should be deleted and created again so that the contain the proper AttributeValues and CharacteristicValues.

3.5 Acceleo Generator

The Acceleo generator is based on an EMF model and a set of templates to generate text files. In the templates the information of an instance of the EMF model is accessed to generate the files.

Templates are like methods, they have a name, parameters with names and types, and visibility (public | protected | private). They can also be called from other templates and a main template has to be specified. The main template is specified by adding the "[comment @main/]" comment. Acceleo documentation advices against having more than one main template.

Inside a template there are two kinds of expressions: static text and Acceleo expressions. Static text is generated without any changes, while Acceleo expressions are used to evaluate elements of the model.

FIX This is done with the Acceleo operation select, that selects the elements that meet a condition written as an OCL expression from an iterable object.

Templates where structured in the following way. There is a main template that takes a CharacteristicComponent object as an argument. This template calls one template for each file to be generated. In this calls, the CharacteriscticComponent object used in the main template (or *self*) is passed by default. All the templates generate from this characteristic component, accessing its attributes and children.

Note that the generator uses the implementation files for the generation so changes in the model can affect the generation even if the templates and the instance is the same. As an example, if a getter is called twice by a container's implementation, the generator will generate it two times, even if the instance's XMI has the value only once.

3.5.1 Template Description

General description of each of the generator's templates.

generateComponent

Main template. Its only function is to call all the other templates.

generateIDL

Generates the IDL file. Uses the component's name to name the interface and create the definition guard, the component's module name to name the module, specifies the prefix if one is set, and iterates through properties and actions to declare them as part of the CORBA object's interface.

generateSchema

Generates the XML Schema file. Uses the component name to specify namespaces, the complex type, and declare it. Also iterates through the properties and attributes in the complex type definition.

generateObjectConfig

Generates the Components configuration XML, where the instances are declared. Iterates through the component instances to get their name, "autostart" and "default" values, and the component's name, container, module and prefix.

generateRTConfig

Generates one file for each instance in separate folders, following the ACS directory structure. The generated file differs slightly when there are no properties defined so this is checked before iterating through the instance's attribute values and characteristic values for each property.

generateImplHead

Generates the header for the C++ implementation. Includes the main baci libraries statically and the specific baci libraries for each used baci type. Also includes the libraries for all used DevIOs and <Component>S.h. Declares properties, actions. smart property pointers, devIO read and read-write objects, and the variables used to construct them. All of these are declaredas part of the <Component>_impl class, and within a declaration guard. Properties are functions with ACS::<baciType>_ptr return type, and actions are normal C++ functions.

Smart property pointers, devIOs and property specific devIO variables are named and declared separately for both access types (RO and RW). This is done by selecting the properties according to the access type of their baci type (A string comparison is necessary because the OCL expression does not interpret the value (0 or 1 integer) as boolean). DevIO variables are also named and declared separately according to the isPropertiSpecific, isRead and isWrite values.

Declares a string named "component_name" statically. This variable is used to get the component name in the constructor and use it in the initialize function. Since the initialization should be done in the initialize function instead of the constructor (according to the BACI Device Server Programming Tutorial) this global variable is required to create smart property pointers.

generateImpl

Generates the C++ implementation file. Includes only the header. Has the base code for all the methods declared in the header. The constructor has a minimal implementation, in which the component_name variable is initialized and an ACS_TRACE with the Comopnen name called. The destructor is left empty.

All the life cycle functions have user code blocks. Initialize and cleanUp have additional generated code. The initialize function has a comment block in which the variables whose initialization must be implemented by the user are shown. After this block, there is the code where the values of the component's attributes are obtained from the CDB. All the attributes are obtained as strings and stored as local variables named <Attribute>_from_cdb. Then there is a user code block where the user has to initialize the variables used to create the devIO objects. The devIO's objects are created to create the smart property pointers. There is one devIO object for each property and it can be read or readWrite depending on the baci type of the property. Smart property pointers are then create using the component_name and the devIOs and then another user code block is generated where additional initialize functionality can be implemented.

In the cleanUp function the devIO objects are deleted.

The properties are implemented in a standard way. They return an ACS:::<baciType>_var._retn() that is obtained calling ACS::<baciType>::narrow with the CORBA reference from the smart property pointer as argument.

Actions only have a user code block because they are considered to be of arbitrary implementation.

generateMakefile

The generated Makefile is mostly copied from a template. The only elements that are used in the generation of this file are the component name and the libraries required by the devIO, which are a string, copied in one line for each used devIO. This generation could definitely be more efficient as most of the file is not used.

4 Future Work

As this project was developed during a summer internship, there are several things to refine and to explore remaining. Most of them aim to make the generator easier to use and maintain, and make the generation more flexible and informative, since the main functionality is complete. The *To do* section talks about changes that are most probably direct improvements while the *To explore* section contains topics that could be looked at as possible improvements.

4.1 To do

- Launch the editor as an Eclipse plugin. This way it should be easier to get running and the meta-model implementation will be transparent to the user.
- Launch Acceleo as a runnable jar. Then the generator could be called from the command line.
- Add ACS logging and debugging messages to the implementation.
- Clean the makefile.
- Have each template in a different file. This would make them easier to inspect and maintain.
- Add comments to each file, describing the generation. At least to let anyone inspecting the code know that is was generated by this generator.
- Check that the names of the model and files are clear to the users and follow conventions.
- In the instance configuration files, only generate the characteristics that are modified by the user. Currently all characteristics for a Property are generated in the run time configuration, even if they have the default value, which is redundant. Could be achieved by having a derived attribute that indicates if the value has been changed.

4.2 To explore

- Add generation information to the EMF model. Some metadata could be added to the generation comments like the author, creation and modification dates, etc.
- Add all features of an XML attribute to the Attribute class. In practice the component attributes are exactly XML attributes, so maybe it would be desirable to model them exactly like that.
- Do some re-factoring to the modifications in the model's Java files. Apparently there are appropriate places where such modifications should be, and in case of using the model for some application other than the editor, this could become relevant. The current implementation works correctly and no direct guidance on the subject was found. This may require a deeper understanding of EMF so it has been left as future work.
- Generate a base implementation for the DevIO with the given naming conventions. This would speed up the implementation of a new DevIO.
- See if its necessary to obtain Attribute values from the CDB in the implementation that are of a type other than string. Current implementation only does it as string types.
- Add containers for Actions, Attributes and Properties. This would be only to make the editor better organized.

5 References

- BACI_Device_Server_Programming_Tutorial.pdf[1]
- ACS_Basic_Control_Interface_Specification.pdf[2]
- ACS_Supported_BACI_Types.doc[3]
- Eclipse Help - Acceleo Documentation[4]
- Acceleo User guide[5]

1 https://confluence.alma.cl/download/attachments/25792175/BACI_Device_Server_Programming_Tutorial.pdf?api=v2&modificationDate=1550601710000&version=1
2 https://confluence.alma.cl/download/attachments/25792175/ACS_Basic_Control_Interface_Specification.pdf?api=v2&modificationDate=1550601777000&version=1
3 https://confluence.alma.cl/download/attachments/25792175/ACS_Supported_BACI_Types.doc?api=v2&modificationDate=1551441174498&version=1
4 https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.acceleo.doc%2Fpages%2Findex.html&cp=5
5 https://wiki.eclipse.org/Acceleo/User_Guide

6 Glossary

The project uses most of the terms as they are in the ACS Documentation but some may have a more specific meaning or differ from the usage in the current implementation.

Attribute - In the context of the Characteristic Component, it refers to Instance specific variables that are available as part of the CDB. It may also refer to an XML attribute.

CDB - *Configuration Data Base* -

COB - *CORBA Object* -

Component - Characteristic Component

DevIO - *Device Input/Output* -

IDL - *Interface Definition Language* - Also used to name the file in which the CORBA interface is defined.

Instance - In the context of the generator, this may refer to the model created with the generated editor. In the context of the component, it refers to a deployment instance of the component.

Meta-model - Generator data model for which an instance must be created to generate the component.

Run time configuration - Initial configuration for the component's instances located in test/CDB/alma/...