University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

11-29-2019

Automated Generation and Integration of AUTOSAR ECU Configurations

Usha Sreeram
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
Sreeram, Usha, "Automated Generation and Integration of AUTOSAR ECU Configurations" (2019). *Electronic Theses and Dissertations*. 8149.
https://scholar.uwindsor.ca/etd/8149

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Automated Generation and Integration of AUTOSAR ECU Configurations

By

Usha Sreeram

A Thesis

Submitted to the Faculty of Graduate Studies

through the Department of Electrical and Computer Engineering

in Partial Fulfillment of the Requirements for

the Degree of Master of Applied Science

at the University of Windsor

Windsor, Ontario, Canada

© 2019 Usha Sreeram

Automated Generation and Integration of AUTOSAR ECU Configurations

by

Usha Sreeram

APPROVED BY:

____

T. Bolisetti

Department of Civil and Environmental Engineering

____

B. Balasingam

Department of Electrical and Computer Engineering

____

M. Khalid, Advisor

Department of Electrical and Computer Engineering

November 27th, 2019

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Automotive Open System Architecture (AUTOSAR) is a system-level standard that is formed by the worldwide partnership of the automotive manufacturers and suppliers who are working together to develop a standardized Electrical and Electronic(E/E) framework and architecture for automobiles. The AUTOSAR methodology has two main activities: system configuration and the Electronic Control Unit (ECU) configuration. The system configuration is the mapping of the software components to the ECUs based on the system requirements. The ECU configuration process is an important part of the ECU software integration and generation. ECU specific information is extracted from the system configuration description and all the necessary information for the implementation such as tasks, scheduling, assignments of the runnables to tasks and configuration of the Basic Software (BSW) modules, are performed. This activity allows the ECU to modify the configuration parameters based on the vendor-specific requirements. Due to the high complexity and redundancy of this process, it has to be supported by different tool-related editors that can automatically generate source files like *.c and *.h for the configuration. In this thesis, we propose a method to automate the ECU configuration process for AUTOSAR. We use configuration templates written in *xtend* programming language along with a BSW generator tool developed at APAG Elektronik. This tool can extract the configuration parameters and automatically generate the required ECU module configuration. The Watchdog module will be used as an example to generate and integrate the ECU configuration. This enables the seamless generation of the software configurations from the system level requirements to the software implementation and therefore ensures consistency, correctness, cost efficiency and reduces the work done by the developer to generate the configuration.

Acknowledgments

With utmost sincerity, I express my gratitude and respect to my advisor Dr. M. Khalid, who gave me the wonderful opportunity to work under his supervision and inspired me to work with honesty, integrity, and discipline.

I would like to thank Mitacs for supporting this work through the Mitacs Accelerate program. Also, APAG Elektronik for funding and supporting my research.

I would also like to thank my committee members Dr. T. Bolisetti and Dr. Balasingam, who provided me with insightful suggestions to improve my research.

I would like to dedicate my work to my parents, as their ever-encouraging faith has kept me going and gave me the strength to overcome any obstacles that have come my way.

Table of Contents

Author's Declaration of Originality iii

Abstract iv

Acknowledgments v

List of Tables viii

List of Figures ix

List of Abbreviations xi

Chapter 1. Introduction 1

1.1. Motivation 1

1.2. Objective 2

1.3. Thesis Outline 2

Chapter 2. Background 4

2.1 AUTOSAR 4

2.2. AUTOSAR Methodology 9

2.3. AUTOSAR Extensible Markup Language 12

2.4. Introduction to Xtend 12

2.5. Related Research 14

Chapter 3. AUTOSAR ECU Configurations 17

3.1. Configuration Classes 18

3.2. Configuration Metamodel 21

3.3. ECUC Parameter Model 22

3.4. ECUC Model 22

3.5. Module Configuration Template (MCT) 25

Chapter 4. Automated Generation of ECU Configurations for Watchdog Timer 26

4.1. Watchdog Timer in AUTOSAR 26

4.1.1. Watchdog Driver Module 26

4.1.2. Watchdog Interface Module 27

4.1.3. Watchdog Manager Module 28

4.2. Module Configuration description 31

4.3. ECUC Model 32

4.4. Module Configuration Template 39

4.5. BSG Tool 47

Chapter 5. Functional Testing and Evaluation 55

5.1. Auto-Generate Source Code 55

5.2. Tests Cases for the BSG tool 56

5.3. Approximate Time and Cost comparisons 64

Chapter 6. Conclusion 65

6.1. Summary 65

6.2. Future Work 66

References 67

Appendix A. AUTOSAR Methodology 4.4 69

Appendix B. Executable Java Code 70

Appendix C. BSG class diagram 72

Vita Auctoris 73

List of Tables

Table 4.1. Wdg module-specific information [11] 27

Table 4.2. WdgIf module specification information [12] 28

Table 4.3. WdgM module-specific information [13] 28

Table 4.4. WdgIfVersionInfoApi sample [11] 31

Table 4.5. ECUC model auxiliary functions 34

Table 4.6. List of auxiliary functions for formatting the output source code 44

Table 4.7. Feedback in the log file 53

Table 5.1. Take Cases 57

List of Figures

Figure 2.1. AUTOSAR Architecture 4

Figure 2.2. Basic Software 5

Figure 2.3. AUTOSAR modules [9] 6

Figure 2.4. Interfaces defined by AUTOSAR [9] 8

Figure 2.5. Overview of AUTOSAR Methodology (version 3.2[16]) 10

Figure 2.6. AUTOSAR development process [16] 10

Figure 2.7. Xtend example – Attributes a – xtend class, b – generated Java code 13

Figure 2.8. Xtend example – Constructors 14

Figure 2.9. Methods 14

Figure 3.1. ECUC process 17

Figure 3.2. Pre-Compile time configuration chain 18

Figure 3.3. Link Time configuration chain 19

Figure 3.4. Post-build time loadable configuration chain 20

Figure 3.5. Post-build time loadable configuration chain 20

Figure 3.6. Parameter definition and ECUC value files 21

Figure 3.7. ECUC parameter model [20] 22

Figure 3.8. Value-Ref Structure 23

Figure 3.9. Class Diagram representing generic module design in AUTOSAR 24

Figure 4.1. WdgM module Configuration [13] 29

Figure 4.2. Watchdog timer file structure 29

Figure 4.3. Sequence flow for the watchdog modules [11] 30

Figure 4.4. arxml module configuration description sample 32

Figure 4.5. ECUC model 33

Figure 4.6. MCT sample for Wdg (Xtend code) 39
Figure 4.7. Complex MCT sample (Xtend code) 41
Figure 4.8. executable Java code 42
Figure 4.9. Module configuration template Workflow 43
Figure 4.10. BSG GUI 47
Figure 4.11. MVC Pattern 48
Figure 4.12. Class diagram for implemented MVC pattern 49
Figure 4.13. BSG tool with loaded config files 50
Figure 4.14. BSG tool flow 52
Figure 5.1. Generate source code Wdg_Lcfg.c 55
Figure 5.2. Generated source code for WdgM_Cfg.c 56
Figure 5.3. example command line 57
Figure 5.4. Approximate time taken to generate ECUC 64
Figure A.1. AUTOSAR Methodology (version 4.4 [16]) 69
Figure A.2. Class diagram for the BSG 72

List of Abbreviations

AUTOSAR	Automotive Open System Architecture
ECU	Electronic Control Unit
E/E	Electrical and Electronic
BSW	Basic Software
CAD	Computer-Aided Design
RTE	Real-time Environment
ECUC	Electronic Control Unit Configuration
MCT	Module Configuration Templates
IDE	Integrated development environment
XML	Xtensible Markup Language
ARXML	AUTOSAR Xtensible Markup Language
BSG	Basic Software Configuration Source Code Generator
SWC	Software Components
OS	Operating Systems
MCAL	Microcontroller Abstraction Layer
API	Application Programming Interface
CDD	Complex Device Driver
I/O	Input Output
AL	Abstraction layer
OEM	Original Equipment Manufacturer

COM	Communication Module
UML	Unified Modeling Language
MDA	Model Driven Architecture
Wdg	Watchdog Module
WdgIf	Watchdog Interface Module
WdgM	Watchdog Manager Module
SEID	Supervised Identity
MVC	Model View Controller
GUI	Graphical User Interface

Chapter 1. Introduction

1.1. Motivation

Computer-Aided Design (CAD) of automotive embedded systems is gaining popularity as a primary design methodology in the automotive industry. It has a number of advantages such as seamless design integration, low cost and reduced development time. It also helps avoid errors and other mistakes that can happen during manual development by a developer. AUTOSAR standard gives the guidelines for the development of Electronic Control Units (ECU) but does not specify the complete design process. Methods and tools need to be developed to automate the process of the ECU development to make it less complex and faster to develop.

There have been a few research efforts in the past that focus on the automated generation of AUTOSAR configurations [1- 4] but they generally focus on generating the Real-time Environment (RTE) or automating different parts of the AUTOSAR methodology. A case study that explains the need for automation within the AUTOSAR ECU configuration (ECUC) process is presented in [1]. MathWorks designed a CAD tool to auto-generate RTE configurations [2]. The complexities in AUTOSAR methodology requires external tools to help simplify the processes of configuration [3]. Some other research studies were focused on automating the process of ECU and RTE configurations using different methods of writing configuration templates and CAD tools [4]. A detailed overview of currently available CAD tools used in the industry is presented in [20-26].

The main motivation for this thesis is to implement Module Configuration Templates (MCT) and a CAD tool to automate the ECUC process based on the AUTOSAR methodology. It can help small automotive suppliers reduce their design costs and time when developing an ECU. It can also reduce the errors that are usually caused by the manual development of ECUs.

1.2. Objective

The main goal of this thesis is to design and implement MCTs and a CAD tool to automate the generation of ECUCs based on AUTOSAR methodology. The MCTs are written using ARXML [5] and Xtend [6] programming languages and a CAD tool using java is developed to generate the output files for the ECUC process. The Eclipse Integrated development environment (IDE) is used for the programming of this project [7]. The main objectives of this thesis are:

- Automate the process of ECU configuration in AUTOSAR.
- Implement the functionality of watchdog timer in the ECU configuration process

The tasks leading to the objectives are:

- Describe the Watchdog module configuration information using ARXML
- Write a template to access values from the ARXML model using xtend programming language.
- Use the module configuration template as input to the BSG tool to generate ECUC source codes.
- Write test cases to test the functionality of the tool and the generated source code.

1.3. Thesis Outline

The remainder of this thesis is organized as follows:

In Chapter 2, the background of AUTOSAR, arxml model description and xtend programming language is briefly described. Also, the related research in this area is discussed. Chapter 3 describes the ECUC process according to the AUTOSAR methodology. It introduces the configuration metamodels, the template structures, and the configuration classes.

Chapter 4 described the implementation of MCTs and CAD tool that can auto-generate the ECUCs of the Basic Software in AUTOSAR. Chapter 5 discusses the results obtained from the implementation and the testing process which is used to validate the results produced by the CAD tool. We finally conclude the thesis in Chapter 6 with a summary and suggestions for future work.

Chapter 2. Background

This chapter provides background information on AUTOSAR, *xtend* and related works.

2.1 AUTOSAR

Automotive Open System Architecture (AUTOSAR) is a system-level standard that is formed by the worldwide partnership of the automotive manufacturers and suppliers who are working together to develop a standardized Electrical and Electronic(E/E) framework and architecture for automobiles. The technical goal of the architecture is to achieve scalability, transferability, reusability, and modularity. AUTOSAR consists of software architecture, methodology templates, conformance testing and application interfaces [8]. As an automotive ECU SW development standard, AUTOSAR is a significant part of the industry. Many OEMs and suppliers consider AUTOSAR as the basis of their development process for designing their ECU architecture and for developing the functionality of the software components. Many researchers, including our research group, are focused on making the standard more efficient and easier to use with the development environment currently used in the automotive industry.

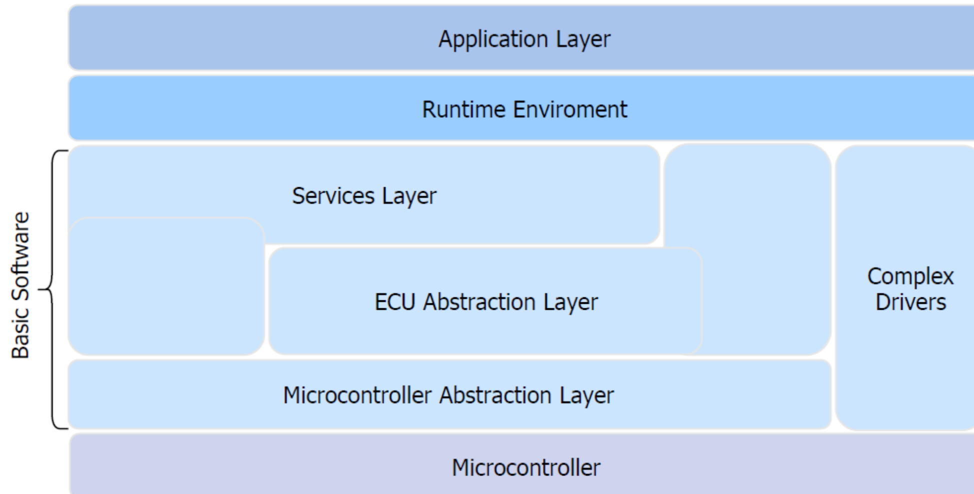

Figure 2.1. AUTOSAR Architecture

The AUTOSAR architecture is divided into three layers: the basic software, the runtime environment, and the application layer, often referred to as AUTOSAR Layered Software Architecture [9]. In Figure 2.1, the first layer is the Application Layer, the second layer is the Runtime Environment (RTE) and last but not least the basic software, which is divided into further layers is shown in Figure 2.1, more on that later. The Application Layer fulfills the functionality of the ECU. It is implemented with the help of one or more software components (SWCs). Here is a distinction between hardware-independent and hardware-dependent SWCs. The hardware-independent ones are called application SWCs, for example, they perform calculations. The hardware-dependent ones are called Actuator or Sensor SWCs. Sensor SWCs can, for example, evaluate signals by debouncing. Actuator SWCs can be used, for example, to control an engine. The next layer, the Runtime Environment, is responsible for facilitating the communication of SWCs among themselves or between SWC and BSW modules including the OS and communication services. For this, the RTE provides the necessary interfaces. RTE is responsible for ensuring that components can communicate and that the system continues to function as expected wherever the components are deployed [10]. The SWCs present in RTE contribute towards the functionality of the AUTOSAR application. AUTOSAR defines standardized interfaces associated with all the SWCs required to develop automotive applications.

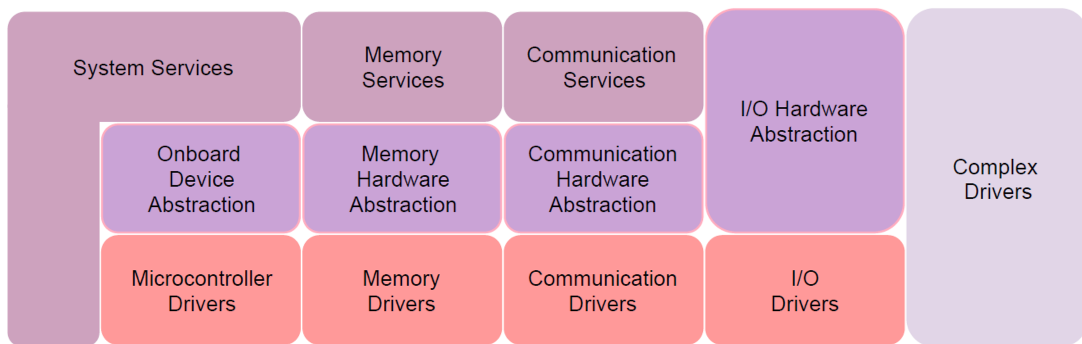

Figure 1.2. Basic Software

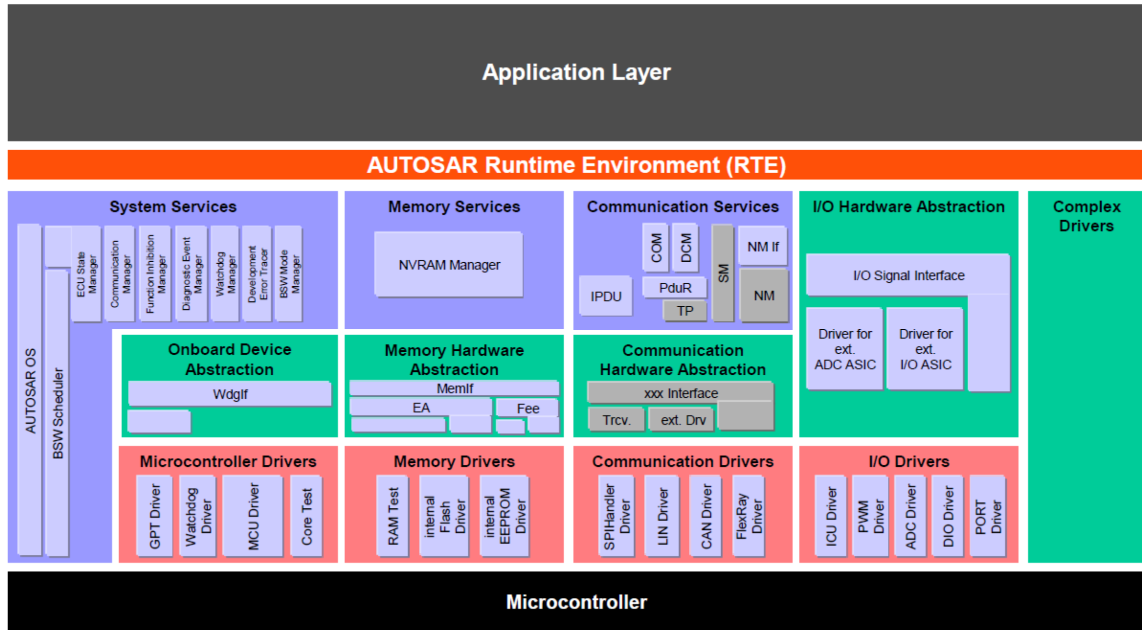

Figure 2.2. AUTOSAR modules [9]

As can be seen from Figure 2.1, The BSW is further divided into complex drivers, microcontroller abstraction layer (MCAL), ECU abstraction layer and service layer. The BSW is responsible for providing services such as operating system functionality, vehicle network communication, memory services, ECU state management, and diagnostics, etc. The service layer is responsible for Operating System (OS) services and, in-vehicle communication, memory services, and diagnostic services. The ECU abstraction layer is hardware dependent and implemented for a specific ECU and offers an Application Programming Interface (API) for access to peripherals and devices regardless of their location on-chip or off-chip and their connection to the microcontroller to make higher software layers independent of the ECU hardware layout. MCAL is dependent on microcontroller and container drivers to enable access to on-chip peripherals. The BSW also contains the Complex Device Driver (CDD) that is used to add a functional model that is outside of the AUTOSAR basic software stack. It provides the option of direct access to the microcontroller via the RTE. The CDD is used only in time-critical functions such as the reaction to a sensor. However, it should be avoided as it undermines the standardized idea of AUTOSAR. The internal structure of the BSW is shown in Figure 2.2. The

arrangement of these layers represents the permitted accesses for RTE to each part as shown in Figure 2.3. Thus, RTE is not allowed to access the Microcontroller Abstraction Layer (MCAL in Figure 2.1) which is also shown by peach-colored parts in Figure 2.2 and Figure 2.3. However, the RTE can access the microcontroller directly via the Complex Drivers. The service layer (Figure 2.1) also shown in purple in Figure 2.2 has the largest connected area to the RTE and, as the name implies, it provides service functions to the application. A part of the ECU Abstraction Layer shown in green in Figure 2.3 is hidden for the RTE. Only the I/O Hardware Abstraction can be accessed by the RTE to abstract the information about the different I/O devices accessed via an I/O signal interface. This thesis deals with the Watchdog modules present across the BSW layers. There are three Watchdog modules in AUTOSAR: Watchdog driver, Watchdog Interface and the Watchdog Manager. The watchdog driver is present in the Microcontroller Abstraction Layer (MCAL) which provides the services for initialization, changing the operation mode and setting the trigger condition for the hardware watchdog [11]. The watchdog interface present in the ECU Abstraction layer (AL) provides uniform access to services of the underlying watchdog drivers like mode switching and triggering [12]. The watchdog manager present in the Services Layer is used to supervise the execution of the ECU program [13].

To ensure the independence and reusability of the software, AUTOSAR defines three different types of interfaces: AUTOSAR Interface, Standardized AUTOSAR Interface, and Standardized Interface. The classification of these interfaces can be traced in Figure 2.4. An "AUTOSAR Interface" defines the information exchanged between software components and/or BSW modules. This description is independent of a specific programming language, ECU or network technology. AUTOSAR Interfaces are used in defining the ports of software-components and/or BSW modules. Through these ports, SWCs and/or BSW modules can communicate with each other (send or receive information or invoke services). AUTOSAR makes it possible to implement this communication between SWCS and/or BSW modules either locally or via a network [14].

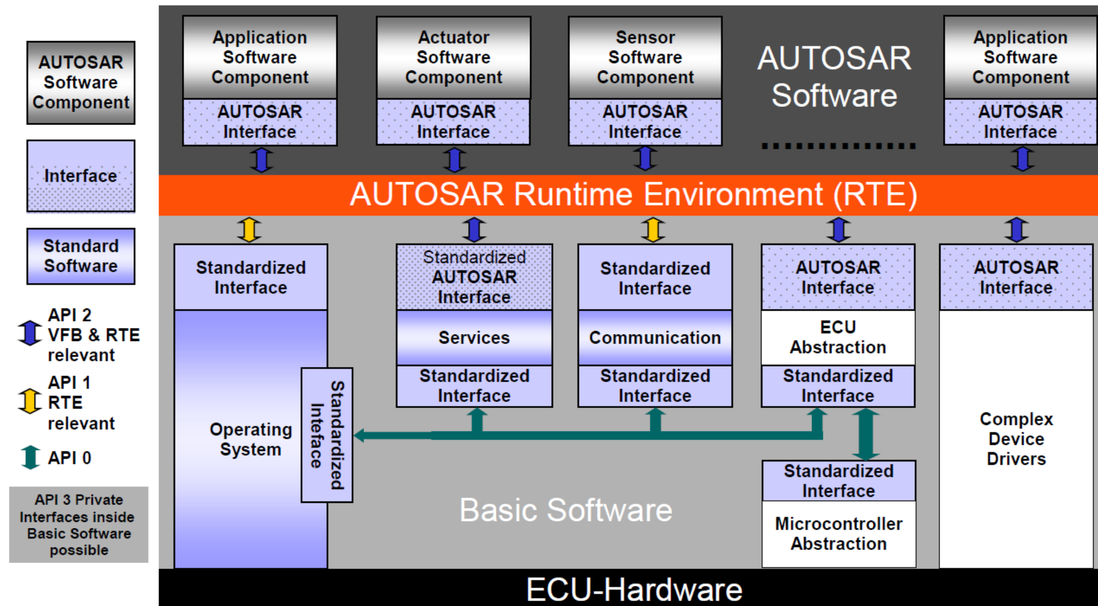

Figure 2.3. Interfaces defined by AUTOSAR [9]

A "Standardized AUTOSAR Interface" is an "AUTOSAR Interface" whose syntax and semantics are standardized in AUTOSAR. The "Standardized AUTOSAR Interfaces" are typically used to define AUTOSAR Services, which are standardized services provided by the AUTOSAR Basic Software to the application SWCs. A "Standardized Interface" is an API that is standardized within AUTOSAR without using the "AUTOSAR Interface" technique. These "Standardized Interfaces" are typically defined for a specific programming language (like "C"). Because of this, "standardized interfaces" are typically used between software modules which are always on the same ECU. When software modules communicate through a "standardized interface", it is NOT possible anymore to route the communication between the software-modules through a network [14][15].

A detailed explanation of the other modules and stacks of AUTOSAR is out of the scope of this thesis.

2.2. AUTOSAR Methodology

This section introduces the AUTOSAR methodology, a condensed overview of that is illustrated in Figure 2.5. AUTOSAR follows a general technical approach to develop a system, which is called as the “AUTOSAR Methodology”. AUTOSAR Methodology is a work-product flow that defines the dependencies of activities on work results.

The first step is to specify initial data for the design or the architecture of the system being developed. That means to select the target hardware ECUs and the software components SWCs. That means to map the modules by regarding timings and resources onto control devices. The result is a “Configuration Description” as AUTOSAR XML file also known as the ARXML file, which contains the complete system information such as bus-mapping, topology, and mapping of containers and parameters. The following steps are processed for each control unit for its own, thus no longer for the complete system. There is a process called Flattening, which is used to generate ECU specific information and this specific information is stored in the “ECU Extract” of System Description. ECU Extract is similar to the System Extract of System Description, but only contains the atomic parameter description in a flat perspective. The following activity “Configuring ECU” feeds all the necessary information such as task scheduling, parameters for basic software modules and allocation of runnables to tasks on the control unit. This information is stored in an “ECUC Description”. In the last step “Generate Executable Code”, a runnable .jar file, which contains the basic software, the RTE code and the software components of the application layer, is generated on the basis of the previously generated “ECUC Description”. Being one of the main goals of AUTOSAR, simplifying integration of application components from the OEM and service providers, the process of application development is independent of the above methodology steps. All the interfaces-related information of SWCs are described in the SWC Descriptions file (AUTOSAR XML file). Based on this description SWCs can be tested and implemented independently. As a result, integration becomes easier. An analysis to understand the data structure of this configuration methodology is done in the next chapter [15].

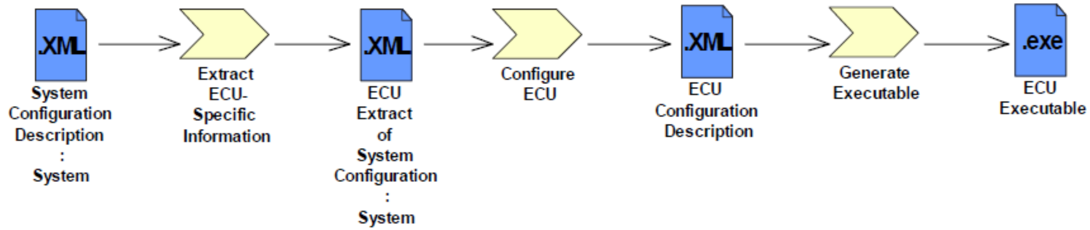

Figure 2.4. Overview of AUTOSAR Methodology (version 3.2[16])

The AUTOSAR standard was first released in 2005, where the structure and the basic architecture was introduced. The standardized modules were described. A simple process was established and tested to validate the use of AUTOSAR in OEMs. In the later versions, the standard was more stabilized and more SWCs and modules were added. The methodology has more complexities. In the latest AUTOSAR version 4.4 [17] which was used for this research, the methodology is very complex and hard to understand. It can be seen in the Appendix A. But, the 4.4 classic standard defines "Roles and Responsibilities" and features all the standardized modules need by the automotive OEM's and their supplier to develop an ECU. This information was not specified in the older versions.

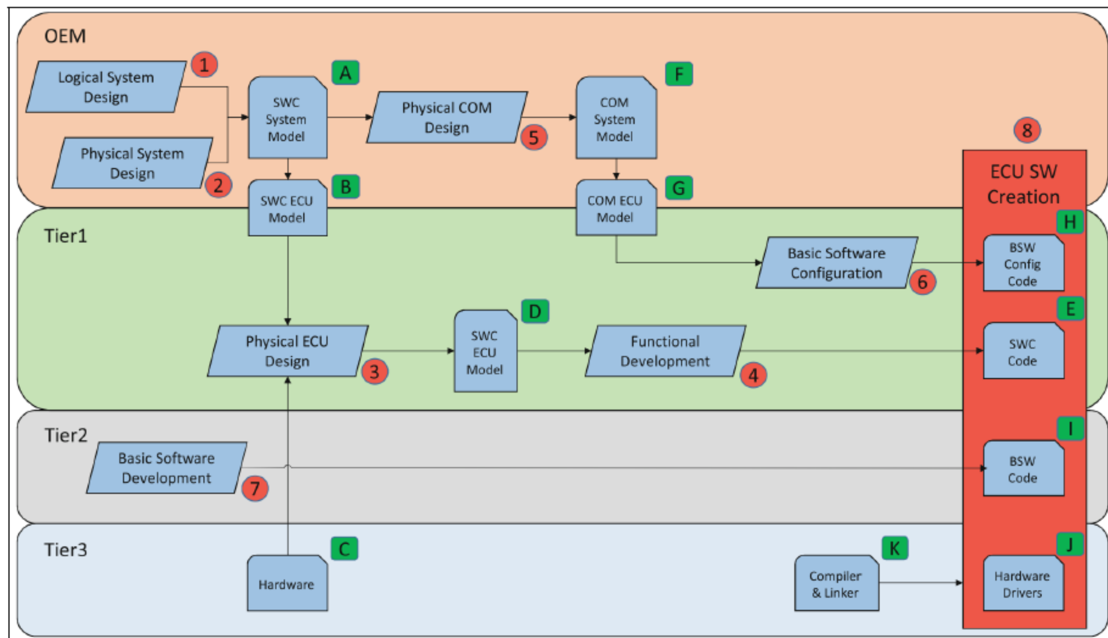

Figure 2.5. AUTOSAR development process [16]

Therefore, the already described Methodology is discussed again with their respective responsibilities. Figure 2.6 shows the individual roles in a hierarchy. Before the task can be divided, the individual roles are briefly presented. There is the OEM, the car manufacturer, who engages suppliers to develop ECUs with the functionality defined by them. There is a classification among the suppliers which describes the scope of the work. There is a distinction between Tier1, Tier2, and Tier3. Tier1 suppliers like Bosch are responsible for the development of the ECU. They develop at the application level of the AUTOSAR architecture. They can hire Tier2 suppliers like APAG who will take care of the BSW development. Tier3 suppliers like Teradyne usually provide the hardware, it does not matter if it is the ECU hardware or just mechanical components or plastic parts. It is also possible for a company to take on several of these roles. APAG Elektronik takes on the role and responsibility of both a Tier 2 and a Tier 3 supplier. Figure 2.6 shows the broken-down process with the responsibilities of each role. The OEM is responsible for the design of the whole system (1 + 2), out of this it provides for each ECU an SWC ECU Model (B), which corresponds to the ARXML File ECU Extract. The relationship between Model and ARXML File is explained in the next section. The SWC System Model (A) thus corresponds to the System Configuration Description. Furthermore, the OEM handles the communication within the system, this is recorded in another model, the COM System Model (F). The communication was not specifically looked at in the previous process. The System Model is also broken down into a COM ECU Model (G). Tier1 receives the SWC ECU Model (B) and the COM ECU Model (G) from the OEM. Based on the hardware, the SWC ECU model is adapted and defined more precisely. Tier2 develops the basic software and makes it available for the application. However, there is no connection to the Basic Software Configuration (6). In the Figure 2.6, this is shown as having only the COM ECU model influence to point 6. There is some kind of BSW Model missing, which is generated from the Basic Software Development and is extended by the SWC ECU Model and COM ECU Model. This BSW model should then be used for the Basic Software Configuration. The proposed BSW Model corresponds to the ECUC Description and will be continued in the next section. [18]

2.3. AUTOSAR Extensible Markup Language

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It was designed to store and transport data across web services. The AUTOSAR Extensible Markup Language (ARXML) is a type of XML language description for exchanging AUTOSAR models and descriptions. The ARXML models are used to represent the AUTOSAR based ECUC. It was formed by an initiative of automotive manufacturers and suppliers to be used by AUTOSAR for the development of the software architectures for ECUs. ARXML files contain configuration and specification information in XML format for an ECU which is used to control components of an automobile to make sure it achieves its optimal performance.

The ARXML format was developed to standardize data exchange between automotive software development partners. These files are integrated into the AUTOSAR stack as per the AUTOSAR methodology. During this process above information gets transformed into AUTOSAR modules as specified in AUTOSAR system specifications [19]. In this project, the ARXML code is the first step of writing the module configurations. This export of the data is from AUTOSAR module configuration templates is called ECU description file which is in .arxml format. This file is used module configuration template to access the parameter information and generate source code specific to the AUTOSAR module which is in our case the Watchdog timer module.

2.4. Introduction to Xtend

Xtend is a statically-typed programming language which translates to comprehensive Java code [20]. *Xtend* is a derivative of Java programming language and it is fully compatible with it. The compiled *xtend* code automatically generates Java source files which are used as input to the BSG tool in this research. *Xtend* is fully compatible with the Eclipse IDE, therefore, it is the best known to create *xtend* projects using Eclipse IDE. Let’s have a look at the following *xtend* examples, on the left side is the *xtend* code and on the right side is the generated java source code.

```
1  class Example {
2    // This comment does not appear in the
         generated .java file
3    /* attributes - This is the first
         visible comment in .java file */
4    var bar = new LinkedList<String>() //
         semicolon not necessary
5    var String stringVar; // default
         visibility for attributes is
         private
6    var package intVar = 3; // visibility is
          package-private
7    var public int intVar2; // visibility is
          public
8    val dar = 'only "a" value'; // ' is used
          for strings as well
9    val s = '!'; // transformed to a String
         instead of char!
10   val char c = 'c'
```

```
1  public class Example {
2    /**
3     * attributes - This is the first
          visible comment in .java file
4     */
5    private LinkedList<String> bar = new
         LinkedList<String>();
6
7    private String stringVar;
8
9    int intVar = 3;
10
11   public int intVar2;
12
13   private final String dar = "only \"a\"
         value";
14
15   private final String s = "!";
16
17   private final char c = 'c';
```

Figure 2.6. Xtend example – Attributes a – xtend class, b – generated Java code

Figure 2.7a shows the different ways how attributes can be declared in *xtend* and Figure 2.7b the generated result is shown. But first of all, it has to be mentioned that there are two kinds of comments. The double slash "//" which are not translated to the generated java class and the slash star "/* */" which are translated to doxygen comments "/** */". *Xtend* doesn't need the semicolon at the end of a command. The data type can be deduced by the initialization. The only differentiation that has to be done is between *var* and *val*. Attributes declared with *val* are only values and can't be changed. Strings are surrounded by double quotes as well by single quotes, because of this feature attention has to be paid by declaring a variable with a character's data type. The default visibility for attributes is private, instead of the Java standard package-private. *Xtend* automatically generates the name of the constructor. Only the keyword *new* is needed, see Figure 2.9. Furthermore, it is possible to break the lines in a string, it is transformed correctly. The default visibility for methods is public. A method's return type need not be defined, see Figure 2.9. To avoid unwanted behavior, it is recommended in most cases to declare

return type because *Xtend* uses the value of the last expression as return type. Other class attributes can be accessed in the *Xtend* source code by using the attribute's name.

```
1  /* constructor */
2  new(int value){
3    bar.add("HELLO")
4    bar.add(new String("W
5 o
6 r
7 l\nd")); // Strings can go over more lines
8    stringVar = '!'
9    intVar = value;
10 }
```

```
1  /**
2   * constructor
3   */
4  public Example(final int value) {
5    this.bar.add("HELLO");
6    String _string = new String("W\r\no\r\
       nr\r\nl\nd");
7    this.bar.add(_string);
8    this.stringVar = "!";
9    this.intVar = value;
10 }
```

Figure 2.7. Xtend example – Constructors

```
1  def foo(){ // default visibility for
       methods is public
2    val jClass = new JClass
3    stringVar = jClass.myVal; // calls the
         getter function
4
5    for( b : bar )
6    {
7      println(b)
8    }
9  }
10
11 def getStringVar(){
12   stringVar // return is not necessary
13 }
14
15 def addValueToBar(String value){
16   bar.add(value);
17 }
```

```
1  public void foo() {
2    final JClass jClass = new JClass();
3    this.stringVar = jClass.getMyVal();
4    for (final String b : this.bar) {
5      InputOutput.<String>println(b);
6    }
7  }
8
9  public String getStringVar() {
10   return this.stringVar;
11 }
12
13 public boolean addValueToBar(final
       String value) {
14   return this.bar.add(value);
15 }
```

Figure 2.8. Xtend example - Methods

2.5. Related Research

The development of automotive embedded systems and the configuration of the basic software are aimed at automating the workflow to improve consistency and reduce the complexity of the software development process using AUTOSAR. The recent research focus is on creating AUTOSAR toolchains and templates based on the AUTOSAR methodology to automate all the processes which can be time and cost-efficient.

Due to the increasing complexity in the last few years, researchers are concentrating their efforts to manage the automation of the development process of the automotive embedded software. To manage this issue AUTOSAR was formed by a group of companies to standardize and improve the complexity management of integrated E/E architectures through increased reuse and exchangeability of SW modules between OEMs and suppliers [21]. The AUTOSAR methodology provides a work-product flow that defines the dependencies of activities on the work-products which is a piece of information or physical entity produced by or used by an activity. But the methodology does not define the overall system-design and process to carry out the configuration process. Automotive suppliers and OEMs are working on setting up toolchains to automate the configuration process.

Various industries are working on the different modules of AUTOSAR to find improvements and make the standard more efficient and easier to implement. A CAD tool was developed based on the AUTOSAR methodology to automate the generation of modules for the customer-specific ECUs. The AUTOSAR process is complex which makes it time-consuming and error-prone. The specifications of all the Watchdog modules are given by AUTOSAR [22]. The watchdog module will be used during the configuration, these specifications provide the internal information of the module such as their type and size. It will also be used to test the functionality of the generated configurations.

The development of the automotive embedded systems and configuration of the basic software and embedded systems have been researched to reduce the complexity and improve the performance of the systems. The authors in [23][24] present the disadvantages of AUTOSAR which shows us that the AUTOSAR configuration process mainly involves manual coding, followed by verification activities such as code inspections and integration tests. Many of these activities lack tool automation, and so involve manual interaction which is error-prone and time-consuming. This complexity is resolved during this research project which makes the AUTOSAR configuration less complex.

The development process of an automotive embedded tool using a seamless architecture is described by the authors in [25]. They use the architecture based on

AUTOSAR which defines all the module specifications, methodology and application interfaces. System configuration is used to establish the configuration process. ARXML (AUTOSAR Extensible Markup Language) files, which contain the module specifications such as the containers and parameters.

In [26] the authors describe an approach for the design of an automotive embedded code generator. More software problems and defects are found due to the increased complexity in automotive development. The authors use an RTE module to design the code generator in the early phases with the help of a predefined process. This approach reduces the redundancy in the code and also saves time through the automated generation. The generated output of the tool is limited to the RTE source code and the application programming interface (API). The configuration of the Basic software modules such as the watchdog module is not focused like in our research.

In [27] the authors describe an approach to bring AUTOSAR concepts like system development, system configuration, timing analysis, and code generation together. They present a meta-model approach to generate the software using the XML schema. An approach to enhance the model-driven system and safety-engineering framework with AUTOSAR aligned software architecture enabling the seamless description of safety criticality systems is presented [28]. A tool bridge to seamlessly transfer artifacts from system development level to software development level is described. The authors have created a tool for the automated generation of Runtime Environment (RTE) configuration in AUTOSAR. They try to generate the configuration files by interfacing approach that establishes an interface between ASW and BSW based on AUTOSAR RTE and then they are mapped into the hardware-specific implementation.

Chapter 3. AUTOSAR ECU Configurations

AUTOSAR has a standard technical approach for the development of the ECUs called the AUTOSAR Methodology. The methodology describes the workflow of design from the system level configuration to the generation of an ECU executable. The result of each step is delivered to the input of the next step in XML format. The ECUC process in one of the major steps of the AUTOSAR methodology. The ECUC process is shown in Figure 3.1.

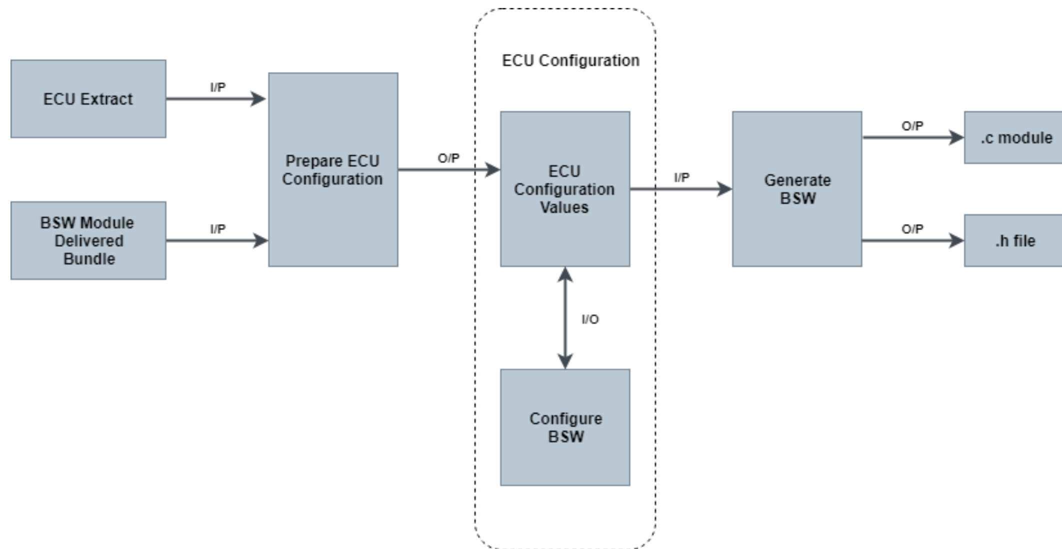

Figure 3.1. ECUC process

The ECUC process starts with the description of an entire system: the system description. This description is then split up into several ECUCs. This ECU extract is the basis for the ECUC process. Every single module of the AUTOSAR architecture can be configured for the special needs of the ECU as specified by the customer requirements. The complex AUTOSAR architecture makes the configuration process difficult and time-consuming. In our research, we use BSG CAD Tool and MCTs to simplify the process. The tool strategy and template details for the ECUC are out of the scope of the AUTOSAR specifications. The tools need knowledge about the ECUC parameters and their constraints such as configuration class, value range, etc [29]. We define this information using an ARXML description that will be used to access the module container and parameter

information. Then an *Xtend* template is written to access the information from the ARXML files and is used at the input to the BSG tool to generate the .c and .h files. Here the configuration parameters are generated into ECU executables which are used to configure the ECU.

3.1. Configuration Classes

The task of compiling and linking is required to create an executable (programmed binary), then the executable must be downloaded (flashed) to the hardware. AUTOSAR specifies for these steps three different times for configuring a BSW module. The "pre-compile time", "link time" and "post-build time". Each of these times has influences on the EcuC Description. The Pre-Compile Time Configuration, shown in Figure 3.2 is done before the compilation. In the case of Pre-Compiled configurations, the code is compiled before the configuration. This can lead to entire code sections can be excluded from the compiled configurations. The advantage that arises is that this way memory space can be saved, and functions will disappear from the compiled files. In order to make the functions available a recompilation of the code is necessary.

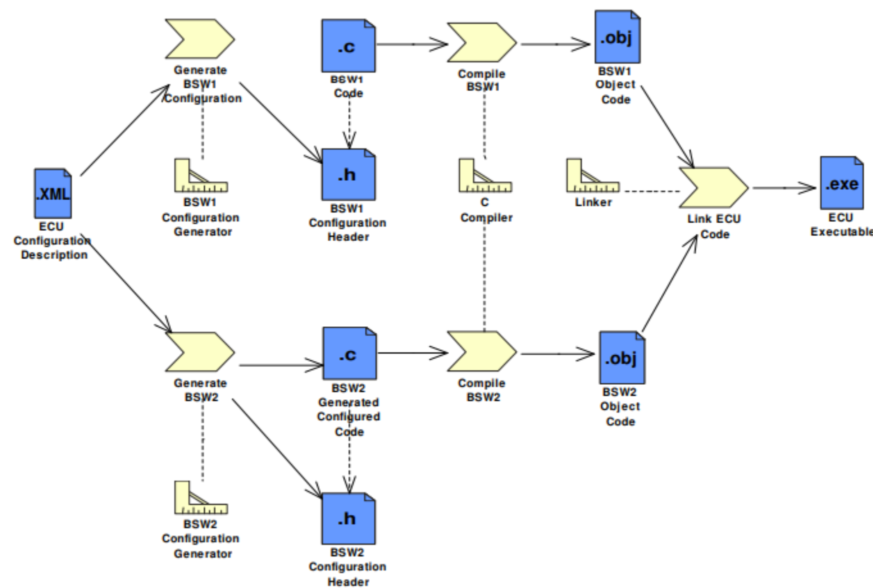

Figure 3.2. Pre-Compile time configuration chain

The Link Time Configuration is created during the link process. Figure 3.3 shows a BSW module consisting of two parts, the code, and the configuration. Both are compiled independently of each other. The object files from the compilation process are linked together, which resolves existing dependencies to external references. After the link process, the values of the configuration cannot be changed anymore.

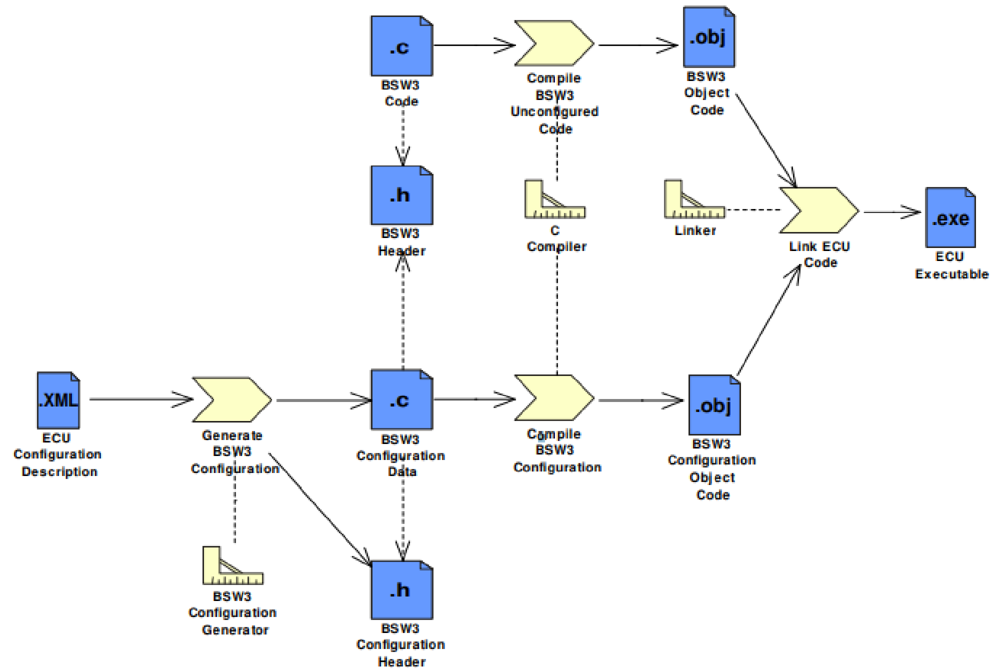

Figure 3.3. Link Time configuration chain

Figure 3.4 shows the Post-Build Time Configuration. The module is already linked and loaded on the ECU. At this point, the module will know the address where the configuration can be found in memory. One risk that this kind of configuration entails is that there is no guarantee that this location in memory has been flashed with the appropriate configuration, if there is a fault in the process, it will not be detected until runtime. For the other two configuration classes, the compiler or linker can ensure that the configuration exists. The advantage of this variant is that the values of the configuration can be changed by rewriting the memory area. AUTOSAR distinguishes between two use cases in the post-build configuration. First, the previously described case shown in Figure 3.4, called loadable configuration and second, the selectable configuration, shown in Figure 3.5. The

process is reminiscent of the link-time configuration but is still considered as post-build. This is because multiple configuration sets are provided at link time. During runtime, more specifically at the initialization of the ECU, one of the existing configurations can be selected. Thus, it can be said that the ECU is configured after building.

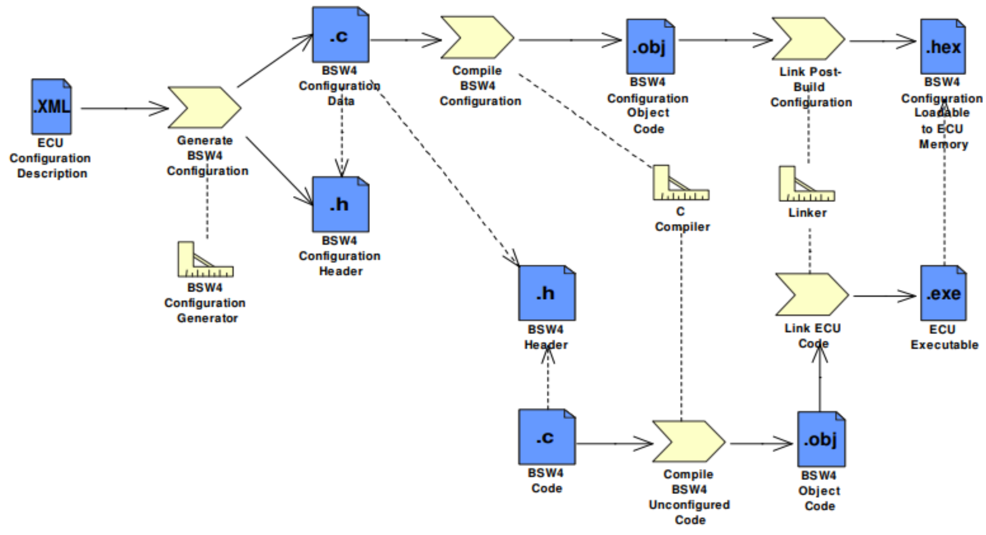

Figure 3.4. Post-build time loadable configuration chain

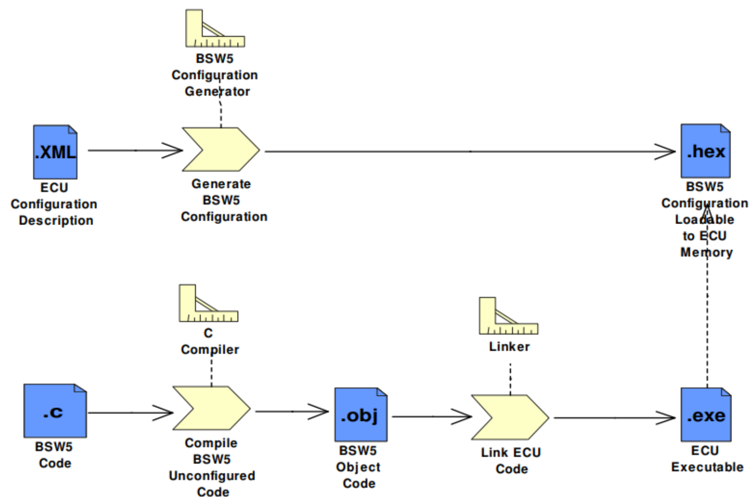

Figure 3.5. Post-build time loadable configuration chain

3.2. Configuration Metamodel

AUTOSAR is based on Model Driven Architecture (MDA). It is a software design approach that uses models for the development of software systems. The model specification is written using Unified Modeling Language (UML). AUTOSAR is made up of several models which are based on metamodels. One of these metamodels is called the Configuration Metamodel. This metamodel describes the structure of the configuration model. It mainly consists of containers and parameters. Containers are used to group the parameters and they can also have sub-containers. The configuration model is used to save the configuration of the BSW, which corresponds to the BSW model mentioned in the previous section. The aim of the metamodel is to make it possible to describe the AUTOSAR specific elements such as the configuration parameters with the same set of language elements. The configuration language generally uses containers and parameters which describe the values that are used to configure the ECU. The configuration metamodel is described in two parts: ECUC parameter definition and ECUC description.

The ECUC description is written using a template to specify the format exchange for the configuration values in the ECU. This template is written using ARXML which was explained in one of the previous sections. The ECUC parameter description which is also an ARXML file contains the information on what kind of restrictions and features are given to the parameters. The relationship between the two is shown in Figure 3.6.

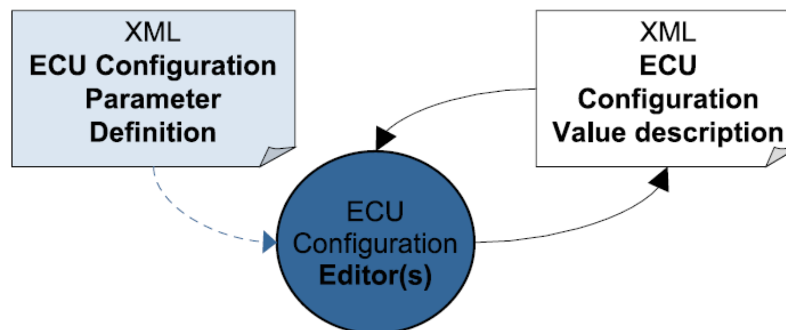

Figure 3.6. Parameter definition and ECUC value files

3.3. *ECUC Parameter Model*

The ECUC parameter model contains the module the information regarding the containers, parameters, and references. It specifies the relationship between the containers and parameters that can be used by the configuration model to describe the module template information. The top-level structure of the ECUC parameter model is shown in Figure 3.7.

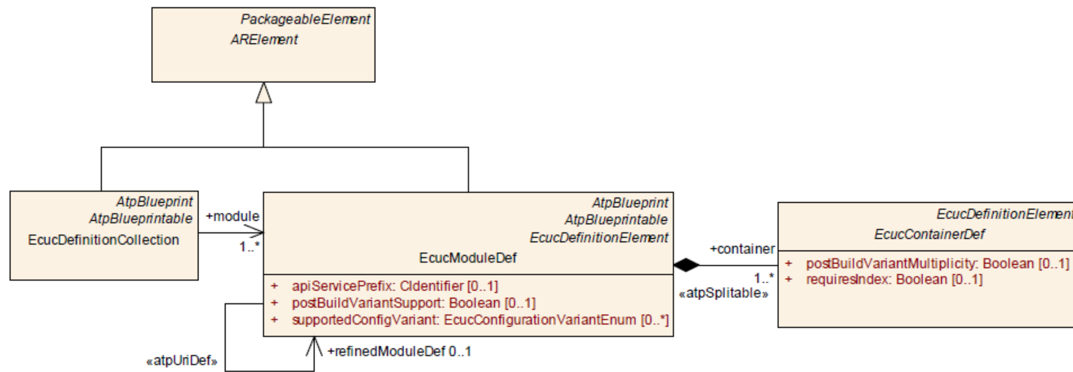

Figure 3.7. ECUC parameter model [20]

ECU Parameter Definition class (EcuParameterDefinition) collects all references to individual module configuration definitions of AUTOSAR ECUC and defines a reference relationship to the definition of several software modules. Module Definition class (ModuleDef) defines ECUC parameters of one software module such as BSW, RTE, SWC.

3.4. *ECUC Model*

This step of the ECUC process is not specified by AUTOSAR. The templates are the most important part of this thesis which helps in automating the configuration process instead of manually configuring the BSW. Figure 3.9 shows the class diagram with the respective attributes. The structure is based on the configuration metamodel. To assist the developer in creating a template, the model provides several auxiliary functions. They are intended to facilitate access to certain elements of the model. For example, the entire configuration of a module should be returned by passing its name. It is distinguished

between name and reference. The *getByName* function searches for a particular name in the model or in a part of the model. Uppercase and lowercase letters are considered. The function *getByRef* uses a reference string to reach the referenced object. An example of a VALUE-REF tag with regard to the class diagram is shown in Figure 3.8. According to the class diagram, we get the following functions:

- getModuleConfigurationByName
- getModuleConfigurationByRef
- getContainerByRef
- getValueByRef
- getArPackageByRef
- getValueByName
- getReferenceValueByValueRefName
- getReferenceValueByDefinitionRefName

```
<VALUE-REF DEST="CONTAINER">/ArPackage.shortName/ModuleConfiguration.shortName/Container.
    shortName</VALUE-REF>
```

Figure 3.8. Value-Ref Structure

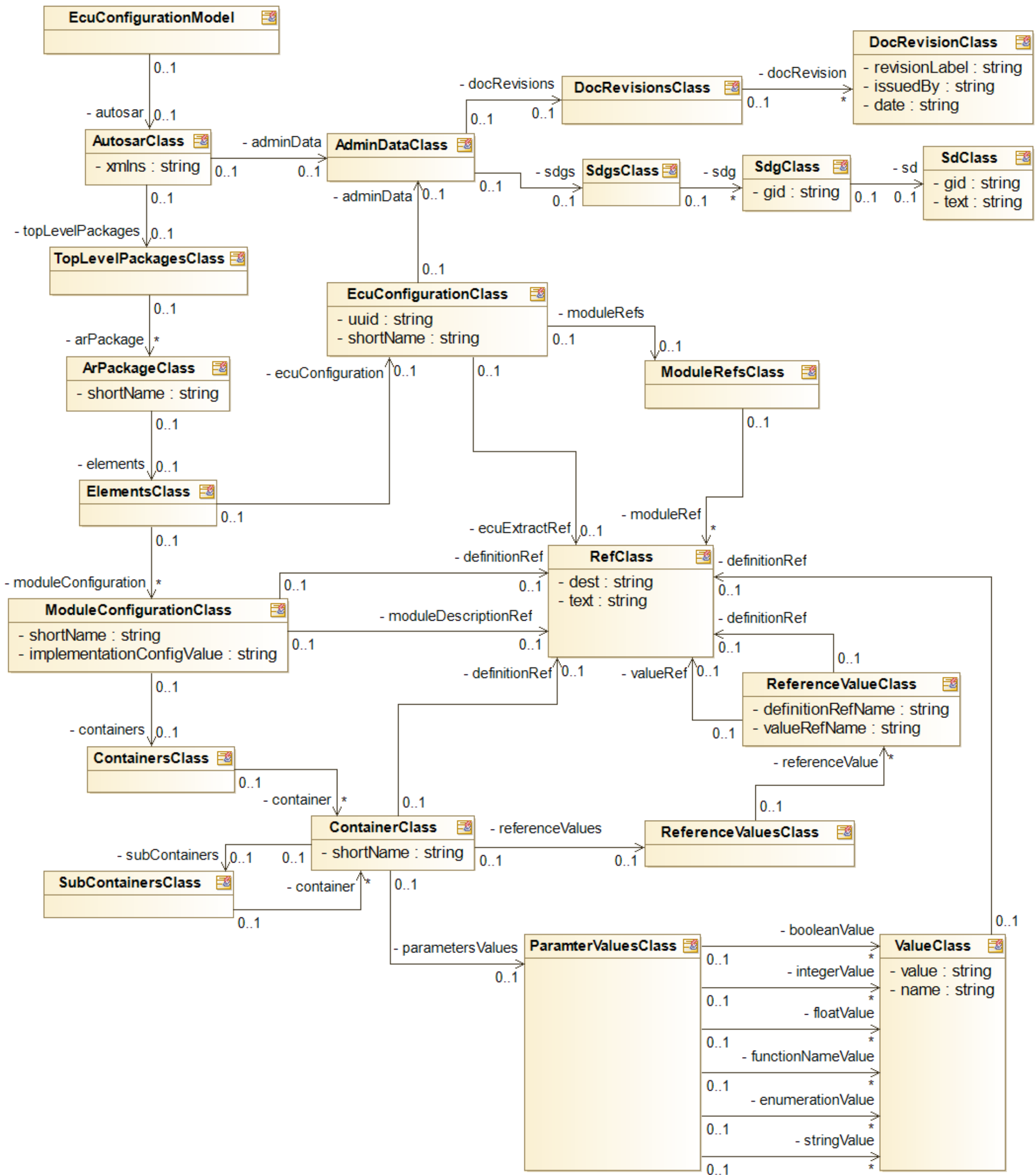

Figure 3.9. Class Diagram representing generic module design in AUTOSAR

A detailed description of these methods can be found in the next chapter. In addition, there are the standard getter functions for getting a class's attribute. The setter functions are not implemented to exclude changes in values and to maintain consistency.

3.5. *Module Configuration Template (MCT)*

The BSW Module Configuration Template provides the template of the source code that should be generated. The values from the ECUC Description can be inserted into the template via the ECUC Model described in the next chapter. The template is written in *Xtend* programming language. The developer of the respective BSW Module is responsible to provide a template for the module configuration. Xtend is used to write the module configuration templates due to its optimized syntax, it allows shorter and readable code and is compatible with Java. The MCT accesses the module values from the ECUC model and describes their function in the ECU. APAG Elektronik provides a library with functions that can be used to access the values without increasing the complexity of the code. These functions are described in the next chapter. Each template comprises a version string. During execution, the version is passed as an argument (-v). Only if the transferred version matches to the template's version, the generation will be executed. Further arguments are the path to ECUC Description (-ecuc) and the path to the output (-o). Without these arguments, the generation won't be started successfully. The Module Configuration Template has to be compiled to an executable and saved with the naming convention "«moduleName»_BswMCT.jar".

Chapter 4. Automated Generation of ECU Configurations for Watchdog Timer

This chapter describes the automated generation of ECUC for Watchdog Timer. The following sections describe the Watchdog Timer in AUTOSAR, implementation of the ECUC model, the module configuration templates and the Basic Software Source Code Generator (BSG) tool.

4.1. Watchdog Timer in AUTOSAR

In order to enable structured software development and to ensure good maintainability, extensibility, and portability of the software, the entire Watchdog Timer module is developed according to a layer model consisting of three layers. the individual layers only communicate with each other via defined interfaces. These interfaces contribute to the security of the software by preventing the individual modules from being able to manipulate any data in any way. In addition, an abstraction of the various layers is achieved, so a layer must (only) know the interfaces to its superordinate or subordinate layer, whereby the complexity of the entire system is encapsulated. The three watchdog modules are:

i. Watchdog Driver (Wdg) – present in the MCAL
ii. Watchdog Interface (WdgIf) – present in the ECU abstraction layer
iii. Watchdog Manager (WdgM) – present in the Services Layer

4.1.1. Watchdog Driver Module

The Wdg Driver module provides services for initialization, changing the operation mode and setting the trigger condition (timeout). This module is used to directly control the hardware watchdog timer and control its function. Table 4.1 shows the Wdg module information with information about the containers that are present in it as specified by AUTOSAR [11]. Each of the containers has parameters that are defined to perform tasks inside the Wdg Module.

Table 4.1. Wdg module-specific information [11]

SWS Item	**ECUC_Wdg_00073 :**
Module Name	*Wdg*
Module Description	Configuration of the Wdg (Watchdog driver) module.
Post-Build Variant Support	true
Supported Config Variants	VARIANT-LINK-TIME, VARIANT-POST-BUILD, VARIANT-PRE-COMPILE

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
WdgDemEventParameterRefs	0..1	Container for the references to DemEventParameter elements which shall be invoked using the API Dem_SetEventStatus in case the corresponding error occurs. The EventId is taken from the referenced DemEventParameter's DemEventId symbolic value. The standardized errors are provided in this container and can be extended by vendor-specific error references.
WdgGeneral	1	All general parameters of the watchdog driver are collected here.
WdgPublishedInformation	1	Container holding all Wdg specific published information parameters
WdgSettingsConfig	1	Configuration items for the different watchdog settings, including those for external watchdog hardware. Note: All postbuild parameters are handled via this container.

4.1.2. Watchdog Interface Module

In case of more than one watchdog device and watchdog driver (e.g. both an internal software watchdog and an external hardware watchdog) being used on an ECU, WdgIf allows the watchdog manager (or any other client of the watchdog) to select the correct watchdog driver - and thus the watchdog device - while retaining the API and functionality of the underlying driver. Table 4.2 shows the WdgIf module information with information about the containers that are present in it as specified by AUTOSAR [11]. Each of the containers has parameters that are defined to perform tasks inside the WdgIf Module.

Table 4.2. WdgIf module specification information [12]

SWS Item	ECUC_WdgIf_00033 :
Module Name	*WdgIf*
Module Description	Configuration of the WdgIf (Watchdog Interface) module.
Post-Build Variant Support	false
Supported Config Variants	VARIANT-PRE-COMPILE

Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
WdgIfDevice	1..*	It contains the information for the selection of a particular Watchdog device in case multiple Watchdog drivers are connected.
WdgIfGeneral	1	This container collects all generic watchdog interface parameters.

4.1.3. Watchdog Manager Module

The WdgM module is used to monitor the sequence for the internal watchdog. This controls whether the 5ms, 10ms, 20ms task was called and processed by the OS. For this purpose, a function with a unique Supervised Identity (SEID) is called at the beginning and at the end of each task. This is to ensure that the corresponding task is called and has completed its activity. The WdgM_MainFunction function regularly checks the process for differences in the task flow. If no error has been detected, the watchdog continues to be triggered normally. In the event of an error, error handling is initiated and the triggering of the watchdog is suspended. Table 4.3 shows the WdgM module information with information about the containers that are present in it as specified by AUTOSAR [13]. Each of the containers has parameters that are defined to perform tasks inside the WdgM Module.

Table 4.3. WdgM module-specific information [13]

SWS Item	ECUC_WdgM_00001 :
Module Name	*WdgM*
Module Description	Configuration of the WdgM (Watchdog Manager) module.
Post-Build Variant Support	true
Supported Config Variants	VARIANT-POST-BUILD, VARIANT-PRE-COMPILE

Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
WdgMConfigSet	1	This container describes one of multiple configuration sets of WdgM.
WdgMGeneral	1	Container defines all general configuration parameters of the Watchdog Manager.

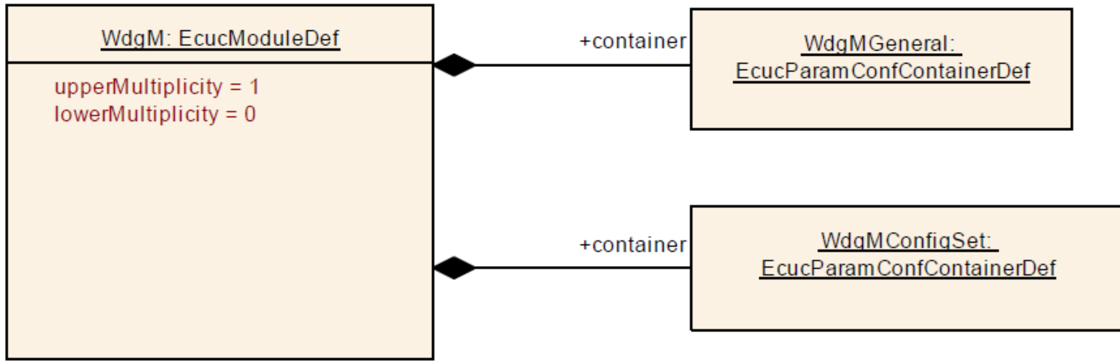

Figure 4.1. WdgM module Configuration [13]

All three watchdog modules work together in AUTOSAR. The file structure is shown in the Figure 4.2. It shows the dependency of the Watchdog modules on each other. The WdgM.c needs to include the Wdg.c, WdgIf.h and WdgM.h files. The WdgM header (WdgM.h) includes the Std_Types.h which is a library containing the type definitions, The WdgM_Lcfg.c and WdgM_Lcfg.h. The Wdg.c includes the Wdg.h file to access the data declarations.

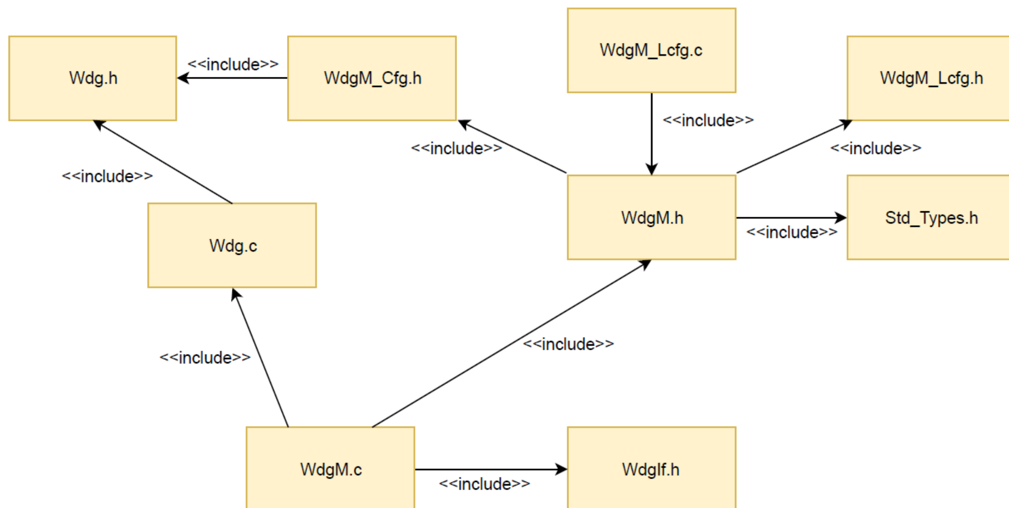

Figure 4.2. Watchdog timer file structure

The sequence diagram shown in the Figure 4.3 shows the sequence of workflow of the watchdog modules. It starts with initializing the Wdg module, to set the trigger

condition and to change the watchdog mode. The Initialization condition to the Wdg is received from the ECU state Manager (EcuM). Once the Wdg is initialized the WdgM sends the trigger condition (*WdgIf_SetTriggerCondition*) to the WdgIf with the information about the Device Index and the Timeout value. The WdgIf selects the right external watchdog timer and sends the trigger condition (*Wdg_SetTriggerCondition*) to the Wdg to trigger the timer and assigns the timeout value. After the watchdog is triggered, the execution of modules begins, each of the modules has a start and endpoint. Once the execution is done the Wdg either needs to be reset or stopped. The SetMode parameter is of the enumerated type that contains the values start, stop and reset. The WdgM also sends out the *WdgIf_SetMode* if there is a need to reset or stop the watchdog timeout counter, which is, in turn, send to Wdg which controls the external watchdog timer. The sequence diagram only shows the basic operation of the watchdog module, the complete functionality of the watchdog timer implemented is not explained due to confidentiality. For more information about the AUTOSAR module specifications and functionality please refer to the AUTOSAR safety documents [10] [11] and [12].

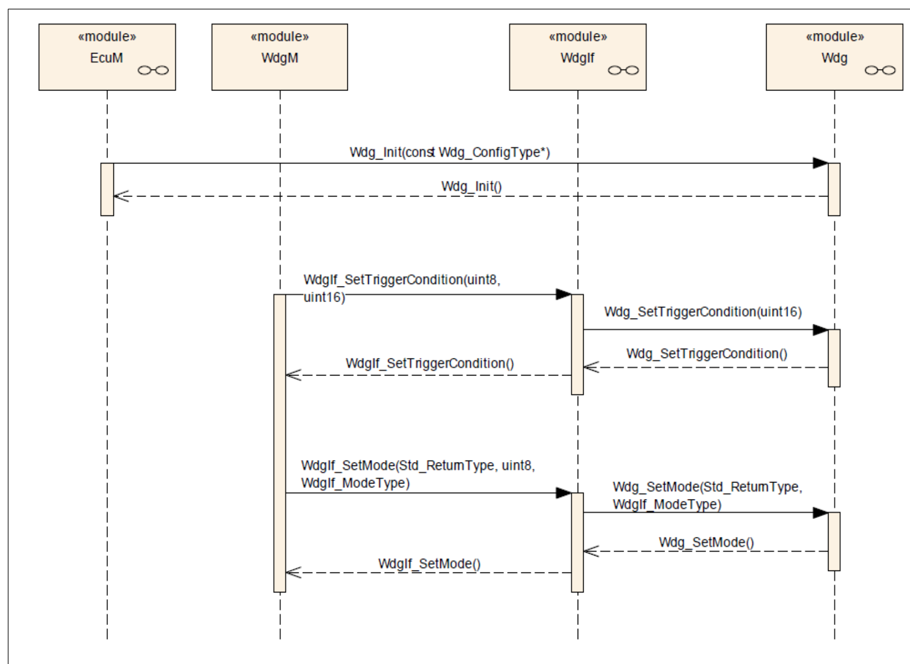

Figure 4.3. Sequence flow for the watchdog modules [11]

4.2. Module Configuration description

The BSW module configuration templates provide the machine-understandable ARXML coded files. It describes the values from classes provided by AUTOSAR module specifications and inserts it into the module configuration template via the ECUC Model. The ARXML model contains all the information regarding the system configuration-related information such as the configuration module name, the containers, the parameters, references and configuration values. The ARXML module configuration description file is used to extract the module information with the help of the ECUC model for the generation of the configuration source codes

Table 4.4. WdgIfVersionInfoApi sample [11]

SWS Item	**ECUC_WdgIf_00005 :**		
Name	WdgIfVersionInfoApi		
Parent Container	WdgIfGeneral		
Description	Pre-processor switch to enable / disable the service returning the version information. true: Version information service enabled false: Version information service disabled false: Version information service disabled		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	FALSE		
Post-Build Variant Value	FALSE		
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	NA	
	Post-build time	NA	
Scope / Dependency	scope: local		

Table 4.4 gives a sample parameter description from the WdgIf module. The ARXML code for the above information is shown in Figure 4.4. All the specifications for the *WdgIfVersionInfoApi* contained between the *PARAMETERS* and */PARAMETERS* tags. This parameter is used to enable or disable the service to return the version information. The parameter is also enclosed between the tag pertaining to its data type *ECUC-BOOLEAN-PARAM-DEF* as mentioned in the specification in Figure 4.4. The configuration class of the module is set to Pre-compile using the tags *CONFIG-CLASS*. The comments in ARXML are written using <!-- -->. The ECUC model extracts the values from this code and it is used by the xtend template to generate the configurations.

```xml
<!-- Container Definition: WdgIfGeneral -->
<ECUC-PARAM-CONF-CONTAINER-DEF>
   <SHORT-NAME>WdgIfGeneral</SHORT-NAME>
      <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
      <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
      <PARAMETERS>
          <!-- PARAMETER DEFINITION: WdgIfVersionInfoApi -->
          <ECUC-BOOLEAN-PARAM-DEF>
              <SHORT-NAME>WdgIfVersionInfoApi</SHORT-NAME>
              <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
              <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
              <IMPLEMENTATION-CONFIG-CLASSES>
                 <ECUC-IMPLEMENTATION-CONFIGURATION-CLASS>
                     <CONFIG-CLASS>PRE-COMPILE</CONFIG-CLASS>
<CONFIG-VARIANT>VARIANT-PRE-COMPILE</CONFIG-VARIANT>
                 </ECUC-IMPLEMENTATION-CONFIGURATION-CLASS>
              </IMPLEMENTATION-CONFIG-CLASSES>
              <ORIGIN>AUTOSAR_ECUC</ORIGIN>
              <SYMBOLIC-NAME-VALUE>FALSE</SYMBOLIC-NAME-VALUE>
          </ECUC-BOOLEAN-PARAM-DEF>
       </PARAMETERS>
```

Figure 4.4. arxml module configuration description sample

4.3. ECUC Model

The ECUC model that was described before is implemented using Java classes. The complete class diagram can be seen in Figure 4.5. This model was developed at APAG Elektronik using UML for this research, it specifies all the necessary functions are needed to write the MCT. The ECUC model uses the ARXML to extract values from the AUTOSAR specification documents. All the auxiliary functions of the model and their functionality are described in Table 4.5. The functions defined under *EcuConfigurationModel* are used to initialize the module using the information from the specifications. The *AutosarClass* functions are used to access module configuration information from predefined AUTOSAR libraries. The values from the Top-level model designs and AUTOSAR Packages are extracted using functions in*TopLevelPackageClass and ArPackageClass* respectively. Similarly, the values from modules, containers and parameters are extracted using the *ModuleConfigurationClass, ContainersClass* and *ParamterValuesClass* auxiliary functions. The detailed explanation of all the functions is given in Table 4.5 and all the functions are mentioned in the class diagram in Figure 4.5.

Some function description is ignored as they are out of the scope of this thesis. Using this information, the MCTs are written which is explained in the next section.

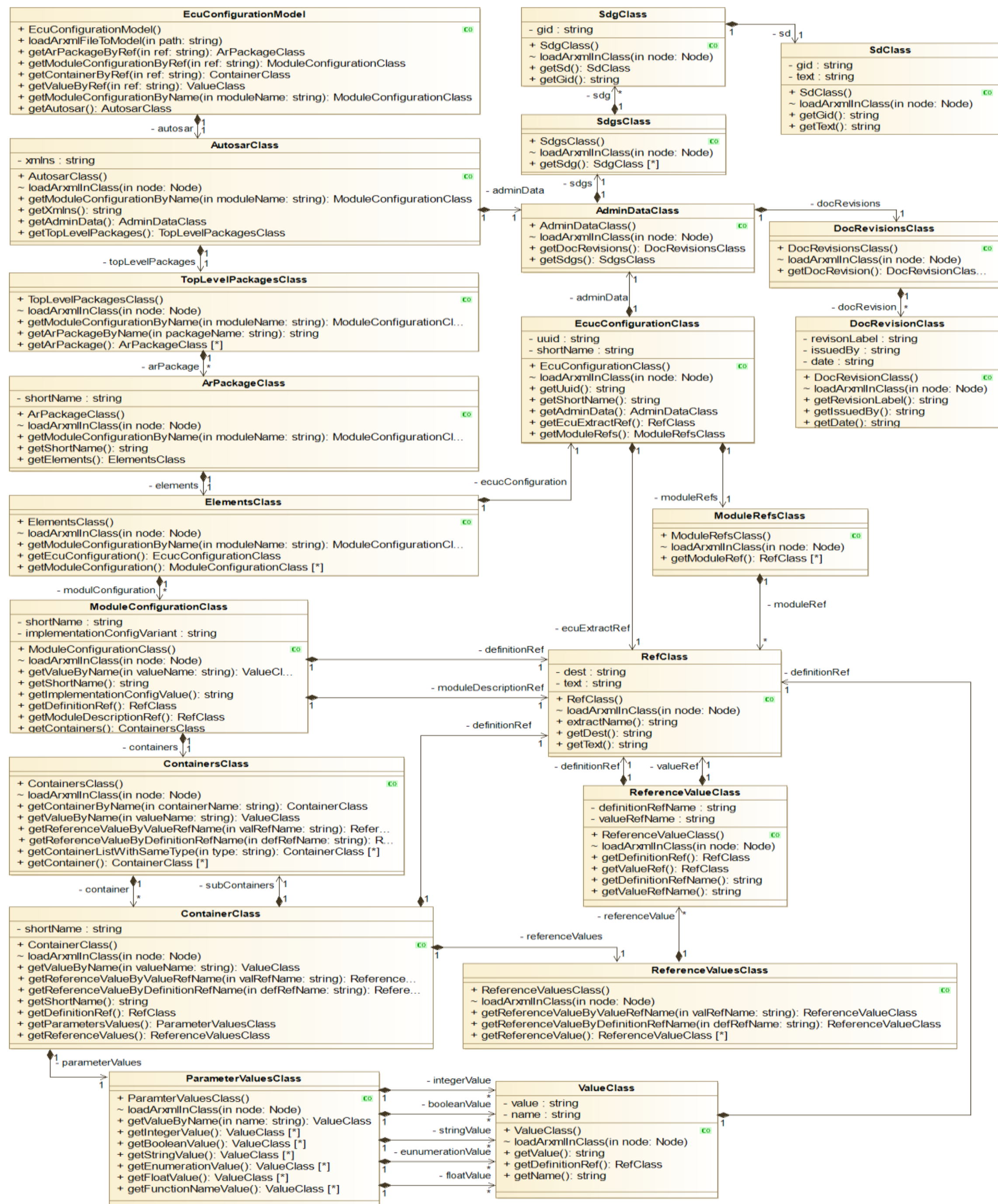

Figure 4.5. ECUC model

Table 4.5. ECUC model auxiliary functions

EcuConfigurationModel	
void **loadArxmlFileToModel** (final String path)	
Description	This function initializes the EcuC model. It calls the nested, package global `loadArxmlInClass` function of the subclasses. It is not expected to merge different EcuC files. An overload is avoided by checking whether the subclass is invalid.
Return	–
ModuleConfigurationClass **getModuleConfigurationByName** (final String moduleName)	
Description	This function is searching a module configuration by its name. It searches through the model to get the `ModuleConfigurationClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ModuleConfigurationClass.shortName) *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.
ModuleConfigurationClass **getModuleConfigurationByRef** (final String ref)	
Description	This function is searching a `ModuleConfigurationClass` object in the model with the reference string.
Note	Expected reference format: /ArPackageName/ModuleConfigurationName
Return	*found*: searched object *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.
ContainerClass **getContainerByRef** (final String ref)	
Description	This function is searching a `ContainerClass` object in the model with the reference string. It also supports the nested search to find a subcontainer.
Note	Expected reference format: /ArPackageName/ModuleConfigurationName/-ContainerName(/SubContainerName...)
Return	*found*: searched object *not found*: an empty (invalid) `ContainerClass` object to avoid null pointer exception.
ValueClass **getValueByRef** (final String ref)	
Description	This function is searching a `ValueClass` object in the model with the reference string. It supports the nested search to find the value in a subcontainer.
Note	expected ref: /ArPackageName/ModuleConfigurationName/Container-Name(/SubContainerName...)/ValueName
Return	*found*: searched object *not found*: an empty (invalid) `ValueClass` object to avoid a null pointer exception.

EcuConfigurationModel

ArPackageClass **getArPackageByRef** (final String ref)

Description	This function is searching an `ArPackageClass` object in the model with the reference string.
Note	expected ref: /ArPackageName
Return	*found*: searched object *not found*: an empty (invalid) `ArPackageClass` object to avoid a null pointer exception.

AutosarClass

ModuleConfigurationClass **getModuleConfigurationByName** (final String moduleName)

Description	This function is searching a module configuration by its name. It searches through the model to get the `ModuleConfigurationClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ModuleConfigurationClass.shortName) *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.

TopLevelPackageClass

ModuleConfigurationClass **getModuleConfigurationByName** (final String moduleName)

Description	This function is searching a module configuration by its name. It searches through the model to get the `ModuleConfigurationClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ModuleConfigurationClass.shortName) *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.

ArPackageClass **getArPackageByName** (final String packageName)

Description	This function is searching an ar package by its name. It searches through the model to get the `ArPackageClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ArPackageClass.shortName) *not found*: an empty (invalid) `ArPackageClass` object to avoid a null pointer exception.

ArPackageClass

ModuleConfigurationClass **getModuleConfigurationByName** (final String moduleName)

Description	This function is searching a module configuration by its name. It searches through the model to get the `ModuleConfigurationClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ModuleConfigurationClass.shortName) *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.

ElementsClass

ModuleConfigurationClass **getModuleConfigurationByName** (final String moduleName)

Description	This function is searching a module configuration by its name. It searches through the model to get the `ModuleConfigurationClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (moduleName equal to ModuleConfigurationClass.shortName) *not found*: an empty (invalid) `ModuleConfigurationClass` object to avoid a null pointer exception.

ModuleConfigurationClass

ValueClass **getValueByName** (final String valueName)

Description	This function is searching a value by its name. It searches through all containers to get the `ValueClass` object whose `shortName` attribute matches to the looked for name. It supports a nested search to find the value in a subcontainer.
Return	*found*: searched object (valueName equal to ValueClass.name) *not found*: an empty (invalid) `ValueClass` object to avoid a null pointer exception.

ContainersClass

ValueClass **getValueByName** (final String valueName)

Description	This function is searching a value by its name. It searches through all containers to get the `ValueClass` object whose `shortName` attribute matches to the looked for name. It supports a nested search to find the value in a subcontainer.
Return	*found*: searched object (valueName equal to ValueClass.name) *not found*: an empty (invalid) `ValueClass` object to avoid a null pointer exception.

ContainerClass **getContainerByName** (final String containerName)

Description	This function is searching a container by its name. It searches through the containers list to get the `ContainerClass` object whose `shortName` attribute matches to the looked for name. It is not searched in subcontainers.
Return	*found*: searched object (containerName equal to ContainerClass.shortName) *not found*: an empty (invalid) `ContainerClass` object to avoid a null pointer exception.

ReferenceValueClass **getReferenceValueByValueRefName** (final String valRefName)

Description	This function is searching a reference value by its value reference name. It searches through all containers to get the `ReferenceValueClass` object whose `valueRefName` attribute matches to the looked for name. It supports a nested search to find the reference value in a subcontainer.
Return	*found*: searched object (valRefName equal to ReferenceValueClass.valueRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.

ContainersClass

ReferenceValueClass **getReferenceValueByDefinitionRefName** (final String defRefName)

Description	This function is searching a reference value by its definition reference name. It searches through all containers to get the `ReferenceValueClass` object whose `definitionRefName` attribute matches to the looked for name. It supports a nested search to find the reference value in a subcontainer.
Return	*found*: searched object (defRefName equal to ReferenceValueClass.definitionRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.

ArrayList<ContainerClass> **getContainerListWithSameType** (final String type)

Description	This function is searching through the containers and lists all containers whose type (definitionRefName) matches to the looked for type, in a new list. The search in the sub containers is not supported, to avoid messing up the order.
Return	*found*: searched object (type equal to ContainerClass.definitionRef.extractName()) *not found*: an empty list of `ContainerClass`

ContainerClass

ValueClass **getValueByName** (final String valueName)

Description	This function is searching a value by its name. It searches through the container to get the `ValueClass` object whose `shortName` attribute matches to the looked for name. It supports a nested search to find the value in a subcontainer.
Return	*found*: searched object (valueName equal to ValueClass.name) *not found*: an empty (invalid) `ValueClass` object to avoid a null pointer exception.

ReferenceValueClass **getReferenceValueByValueRefName** (final String valRefName)

Description	This function is searching a reference value by its value reference name. It searches through the container to get the `ReferenceValueClass` object whose `valueRefName` attribute matches to the looked for name. It supports a nested search to find the reference value in a subcontainer.
Return	*found*: searched object (valRefName equal to ReferenceValueClass.valueRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.

ContainerClass	
ReferenceValueClass **getReferenceValueByDefinitionRefName** (final String defRefName)	
Description	This function is searching a reference value by its definition reference name. It searches through the container to get the `ReferenceValueClass` object whose `definitionRefName` attribute matches to the looked for name. It supports a nested search to find the reference value in a subcontainer.
Return	*found*: searched object (defRefName equal to ReferenceValueClass.definitionRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.

ReferenceValuesClass	
ReferenceValueClass **getReferenceValueByValueRefName** (final String valRefName)	
Description	This function is searching a reference value by its value reference name. It searches through all reference values to get the `ReferenceValueClass` object whose `valueRefName` attribute matches to the looked for name.
Return	*found*: searched object (valRefName equal to ReferenceValueClass.valueRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.
ReferenceValueClass **getReferenceValueByDefinitionRefName** (final String defRefName)	
Description	This function is searching a reference value by its definition reference name. It searches through all reference values to get the `ReferenceValueClass` object whose `definitionRefName` attribute matches to the looked for name.
Return	*found*: searched object (defRefName equal to ReferenceValueClass.definitionRefName) *not found*: an empty (invalid) `ReferenceValueClass` object to avoid a null pointer exception.

ParamterValuesClass	
ValueClass **getValueByName** (final String name)	
Description	This function is searching a value by its name. It searches through all values to get the `ValueClass` object whose `shortName` attribute matches to the looked for name.
Return	*found*: searched object (valueName equal to ValueClass.name) *not found*: an empty (invalid) `ValueClass` object to avoid a null pointer exception.

4.4. Module Configuration Template

As mentioned earlier, *Xtend* is used for the implementation of a Module Configuration Template (MCT). The MCTs are developed to comply with MISRA and ASPICE standards for automotive software development. The templates can be as complex as the developer likes. The Wdg and WdgIf templates are fairly simple but the complexity increases with the Watchdog Manager as it deals with services and interacts with many other modules. A simple sample of the code is given in Figure 4.6 to show how the templates are written to access the values from the ECUC model shown in Figure 4.5.

```
«var WdgGeneralVar = moduleCfg.containers.getContainerListWithSameType("WdgGeneral")»
«FOR var1 : WdgGeneralVar»
const Wdg_ConfigType wdg_initialConfiguration_s =
{
        «var1.getValueByName("Wdg_SetWindowOpenTimeType").value», /* Window open period
        in percent */
        «var1.getValueByName("Wdg_SetErrorModeType").value»,
        /* FEH handle the Error */
        «var1.getValueByName("Wdg_SetIntRequestType").value»,
        «var1.getValueByName("Wdg_SetOverflowIntervalTimeType").value»,
};
«ENDFOR»
```

Figure 4.6. MCT sample for Wdg (Xtend code)

The code in Figure 4.6 shows the definition of the *WdgGeneral* container. The container is accessed using the function getContainerListWithSameType which extracts the container from the ARXML file. Once the container is extracted, the values of the parameters defined for the container are accessed using the ParameterValueClass function *getValueByName.* Everything shown in blue is printed in the generated file as it is seen. The identifier value is used to access the value for the mentioned parameter and print it in the generate file. This is a simple example that shows how a container and its parameters can be accessed.

The second sample in Figure 4.7 shows a very efficient but more complex method that can be very beneficial for arrays or array structures with many entries. The procedure is to create a list of all containers which belong to the same type, this is needed because sometimes a container needs to be created more than once. For example, when there are

more than one hardware watchdog timers present, more than one container with the same name is needed. Therefore, the function getContainerListWithSameType() from the ContainerClass is called. The container name WdgMSeid is compared with the container descriptions in ARXML to find and access the container. An iteration is performed using for loop to in order to access the containers multiple times for every WdgMSeid container that is created. When a Boolean parameter is value is accessed, the value needs to be converted to uppercased as it is described in ARXML to avoid any conflicts that can occur due to case sensitivity. The first two parameters declared are of type Boolean. This sample code also uses some references inside the container. References in AUTOSAR are like pointers, they point to other modules that might contain the information needed by the WdgM. In Figure 4.7, we use the reference *CheckpointRef* to get the reference to the execution start point of every module used in the ECU. The reference path is specified in the ARXML file which will be accessed by the ReferenceClass function getValueRefName(). The other reference used is the *TransitionRef* that is used to point to the reference path where the execution needs to transition from one module to another without completing the execution of the first module. This reference is important when a task calls another function while already executing one. Further details on this topic are not mentioned as they are out of the scope of this thesis. A special feature of xtend is that the template code block is written in between triple single quotes (‴ ‴). The way the statements are written in xtend which you can see in Figure 4.7 is called Lambda Expressions. A lambda expression is basically a piece of code wrapped into an object to pass it around [18]. We can think of a lambda expression as an anonymous class with a single method. The first few lines of code in Figure 4.7 are an example of Lambda expressions.

As the xtend code is written in Eclipse, an executable Java code is generated using the ECUC model. This java file is used as input to the BSG tool to automate the generation of the ECUC. A sample of the java code that is generated by the eclipse is shown in Figure 4.8. The complete Java code for a module is given in the Appendix B.

```
'''
««« WdgM_SEID container definition

«var WdgMSEIDVar = moduleCfg.containers.getContainerListWithSameType("WdgMSeid")»
const WdgM_SEIDType WdgM_SEID_as[WDGM_NUM_OF_SEIDS] =
{
        «var z = 1»
        «FOR SEID : WdgMSeidVar»
        {
                /*.WdgM_SEIDId_ui8 = */  WDGM_REF_ID_LIN«SEID.getValueByName("WdgMSEIDId").value»,
                «IF  SEID.parametersValues.booleanValue.get(3).getValue().equals("true")»
                /* .WdgM_SEIDUsed_b = */
                «SEID.parametersValues.booleanValue.get(3).getValue().toUpperCase()»,
                «ELSEIF SEID.parametersValues.booleanValue.get(3).getValue().equals("false")»
                /* .WdgM_SEIDUsed_b = */
                «SEID.parametersValues.booleanValue.get(3).getValue().toUpperCase()»,
                «ENDIF»
                /* .WdgM_InitState_e = */«SEID.getValueByName("WdgM_InitMode").value»,
                «IF  SEID.parametersValues.booleanValue.get(1).getValue().equals("true")»
                /* .WdgM_Start_b = */
                «SEID.parametersValues.booleanValue.get(1).getValue().toUpperCase()»,
                «ELSEIF SEID.parametersValues.booleanValue.get(1).getValue().equals("false")»
                /* .WdgM_Start_b = */
                «SEID.parametersValues.booleanValue.get(1).getValue().toUpperCase()»,
                «ENDIF»
                «var ref =  SEID.getReferenceValueByDefinitionRefName("WdgMCheckpointRef")»
                «IF SEID.referenceValues.referenceValue.size > 0»
                /* .WdgM_CheckpointRef_ui8 = */    ECUM_Module_«ref.getValueRefName()»,
                «ELSEIF ref.getValueRefName().equals("invalid")»
                /* .WdgM_CheckpointRef_ui8 = */  0, /* Module missing */
                «ENDIF»
                «var icuref = SEID.getReferenceValueByDefinitionRefName("WdgMTransitionRef")»
                «IF icuref.getValueRefName().equals("invalid")»

                /* .WdgM_TransitionRef_ui8 = */         0, /* TransitionRef missing */
                «ELSEIF SEID.referenceValues.referenceValue.size > 0»
                /* .WdgM_TransitionRef_ui8 = */  ECUM_Transition_«TransitionRef.getValueRefName()»,
                «ENDIF»
                /* .WdgM_Access_s = */
                &«SEID.getSubContainers().getContainer().get(0).getShortName()»WdgM«z»
        },
        «{z = z+1; null}»
«ENDFOR»
};
'''
```

Figure 4.7. Complex MCT sample (Xtend code)

Each template needs a Java main for the execution. The Java code shown in Figure 4.8 is used for this purpose. This code is auto generated by Eclipse using the ECUC model. The structure of the java file is the same for all the modules. Only some information needs to be filled by the developer like the moduleName. The information that needs to be filled by the developer is usually marked with the comment TODO. The Java file includes all the ECUC model classes that are used by extend to access the values. The respective module package and java libraries are imported.

```java
package com.apagcosyst.genWdgIf;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import com.apagcosyst.generator.ArgumentsInterpreter;
import com.apagcosyst.generator.IGenSourceCode;
import com.apagcosyst.ecuCModel.ArxmlFileInterpreterClass;
import com.apagcosyst.ecuCModel.EcuConfigurationModel;
import com.apagcosyst.ecuCModel.ModuleConfigurationClass;

public class WdgIf_Main {
	private String moduleName;
	private String moduleVersion;
	private ModuleConfigurationClass moduleCfg;
	private EcuConfigurationModel ecuConfig;
	private IGenSourceCode moduleGen;
	private FileWriter fwCfgh;

	/* insert the configuration for the module which has to be generated */
	public  WdgIf_Main()
	{

		/* TODO: insert here the module name */
		moduleName = "WdgIf";

		/* this is filled in Setup() function */
		moduleCfg = new ModuleConfigurationClass();
		ecuConfig = new EcuConfigurationModel();
		moduleGen = new WdgIf();
		moduleVersion = new String("1.0");

	}
```

Figure 4.8. executable Java code

Code to access such as module configuration class, ECUC model, the module being generated and the module version are automatically generated by Eclipse using the Setup() function. The java executable file contains more statements to access the values from the ECUC model which as mentioned before is shown in Figure 4.5.

The flowchart in Figure 4.9 shows the general behavior of the program and its messages. At first, the arguments are checked, if the correct arguments are not entered, the execution would be canceled. The next step validates the version string, by comparing the transferred version with the version defined in the template. Later, the ECUC Description is loaded to the model and the module configuration of the module is searched for. This information is transferred to the *Xtend* class, which generates the source code by inserting the values into the template.

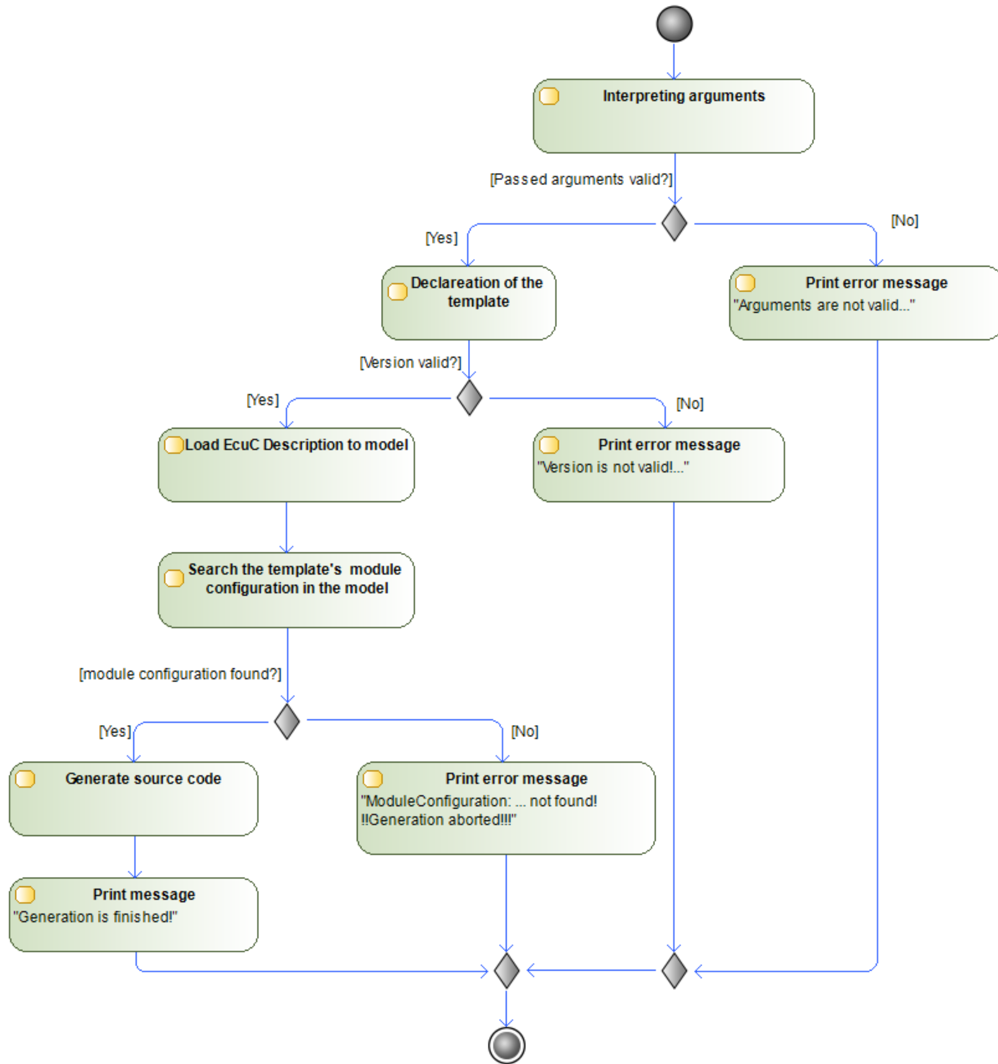

Figure 4.9. Module configuration template Workflow

After the implementation, a Java Archive (JAR) runnable has to be created. This is done in the Eclipse Modeling Framework IDE by a right-click on the project in the "Model Explorer" and select Export. A new window opens, choose there Java > Runnable JAR file and press Next. In the next view, the Launch configuration has to be chosen. Select «moduleName»_Main. Browse for the output folder and name the output «moduleName»_BswMCT.jar. Last but not least select the item "Package required libraries into generated JAR" and press Finish. The runnable is ready for executing. Make

sure the arguments ecuc, -o, -v are passed. Without these arguments, the generation won't be started correctly, as it can be back-traced in Figure 4.9.

Table 4.6 lists the auxiliary functions used to format the output source code. This class also has attributes that can be used for a more generic way, for example, to print large arrays without much effort. In all the functions mentioned, the parameter's value is checked for validity. If the value is not found or is the value found is unequal to the compared string, the result is "invalid".

Table 4.6. List of auxiliary functions for formatting the output source code

printDefineExpression	
static *String* **printDefineExpression** (String name, String value, String casting, String comment)	
Description	This function prints a define expression. It defines the output format for the other `printDefineExpression` methods. The *casting* and *comment* are optional. If the strings are empty they are not printed.
Return	String in the format: `#define` *name* `([(`*casting*`)]`*value*`)` `[/*` *comment* `*/]`
static *String* **printDefineExpression** (String name, ValueClass arValue, String casting, String comment)	
Description	This function checks if *arValue.value* is valid. If it is invalid a warning is shown in the error stream and a preprocessor error string will be returned. In the future the boundaries can be checked and the default value can be used as soon as the information of the *EcuC Parameter Definition* is available. This function calls `printDefineExpression(String, String, String, String)` to format the output.
Return	*valid*: return of `printDefineExpression(String, String, String, String)` *invalid*: `#error: invalid value for parameter` *name*
printParameterExpression	
static *String* **printParameterExpression** (ValueClass arValue, String comment)	
Description	This function casts the value to the correct type. For this functionality the boundaries of the *EcuC Parameter Definition* are needed, at the moment it cannot be used. This function calls `printParameterExpression(ValueClass, String, String)` to automatically generate the variable's name.
Return	return of `printParameterExpression(ValueClass, String, String)`

printDefineExpression

static *String* **printDefineExpression** (ValueClass arValue, String casting, String comment)

Description	This function converts the *arValue.name* in a standardized define name. With this functionality it will be possible to use the define names in a reference. The previous methods allows the developer to chose the name, so it is not possible to make a consistent naming. This function calls `printDefineExpression(String, ValueClass, String, String)` to check the arValue.value.
Return	return of `printDefineExpression(String, ValueClass, String, String)`

static *String* **printDefineExpression** (ValueClass value, String comment)

Description	This function casts the value to the correct type. For this functionality the boundaries of the *EcuC Parameter Definition* are needed, at the moment it cannot be used. This function calls `printDefineExpression(ValueClass, String, String)` to automatically generate the define name.
Return	return of `printDefineExpression(ValueClass, String, String)`

printParameterExpression

static *String* **printParameterExpression** (String name, String value, String casting, String comment)

Description	This function prints a parameter expression. It defines the output format for the other `printParameterExpression` methods. The *casting* and *comment* are optional. If the strings are empty they are not printed.
Return	*name* `=` [`(`*casting*`)`]*value*`;` [`/*` *comment* `*/`]

static *String* **printParameterExpression** (String name, ValueClass arValue, String casting, String comment)

Description	This function checks if *arValue.value* is valid. If it is invalid a warning is shown in the error stream and a preprocessor error string will be returned. In the future the boundaries can be checked and the default value can be used as soon as the information of the *EcuC Parameter Definition* is available. This function calls `printParameterExpression(String, String, String, String)` to format the output.
Return	*valid*: return of `printParameterExpression(String, String, String, String)` *invalid*: `#error: invalid value for parameter` *name*

static *String* **printParameterExpression** (ValueClass arValue, String casting, String comment)

Description	This function uses the *arValue.name* for the naming of the variable. This function calls `printParameterExpression(String, ValueClass, String, String)` to check the arValue.value.
Return	return of `printParameterExpression(String, ValueClass, String, String)`

printParameterExpressionInStruct

static *String* **printParameterExpressionInStruct** (String name, String value, String casting, String comment)

Description	This function prints a parameter expression inside a structure. It defines the output format for the other `printParameterExpressionInStruct` methods. The *name*, *casting* and *comment* are optional. If the strings are empty they are not printed.
Return	[`/* .`*name* `= */`] [`(`*casting*`)`]*value*`,` [`/*` *comment* `*/`]

static *String* **printParameterExpressionInStruct** (String name, ValueClass arValue, String casting, String comment)

Description	This function checks if *arValue.value* is valid. If it is invalid a warning is shown in the error stream and a preprocessor error string will be returned. In the future the boundaries can be checked and the default value can be used as soon as the information of the *EcuC Parameter Definition* is available. This function calls `printParameterExpressionInStruct(String, String, String, String)` to format the output.
Return	*valid*: return of `printParameterExpressionInStruct(String, String, String, String)` *invalid*: `#error: invalid value for parameter` *name*

static *String* **printParameterExpressionInStruct** (ValueClass arValue, String casting, String comment)

Description	This function uses the *arValue.name* for the naming of the variable. This function calls `printParameterExpressionInStruct(String, ValueClass, String, String)` to check the arValue.value.
Return	valid: return of `printParameterExpressionInStruct(String, ValueClass, String, String)`

static *String* **printParameterExpressionInStruct** (ValueClass arValue, String comment)

Description	This function casts the value to the correct type. For this functionality the boundaries of the *EcuC Parameter Definition* are needed, at the moment it cannot be used. This function calls `printParameterExpressionInStruct(ValueClass, String, String)` to automatically generate the variable's name.
Return	return of `printParameterExpressionInStruct((ValueClass, String, String)`

4.5. BSG Tool

The tool responsible for managing and executing the templates to generate the configuration files is called the Basic Software Configuration Source Code Generator (BSG) tool. The tool uses the Graphical User Interface (GUI) to interact with the user. The GUI of the BSG is shown the Figure 4.10. The GUI allows the user to set up the tool to generate the *.c and *.h files for the specific module.

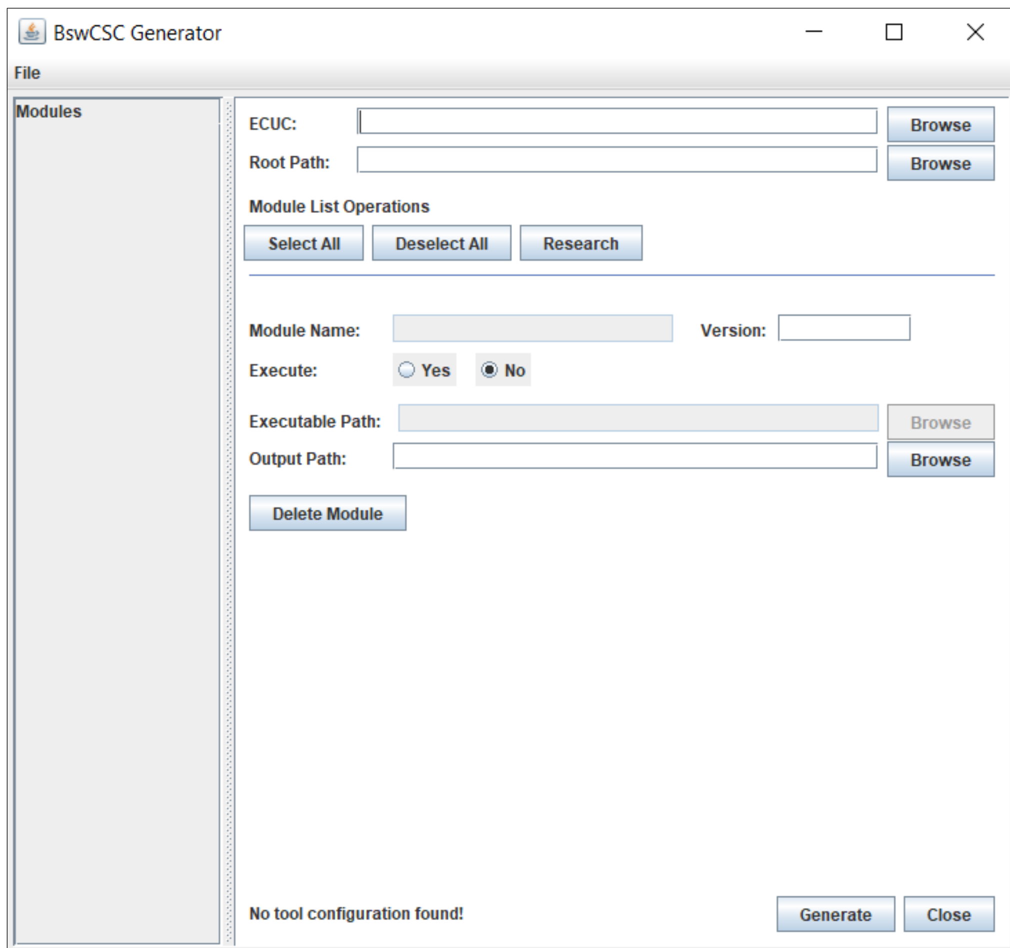

Figure 4.10. BSG GUI

Functions such as the automatic search for MCTs and the ARXML files within the project are implemented. The functionality requires a naming convention of the templates and also a root path for searching the templates. All the existing MCTs are listed on the left when the BSG is opened. Each module consists of the properties such as module name, whether it should be generated, the path and name of the template, the output path, and the version. The default configuration properties for a module are loaded when the module is selected. The properties listed are:

- *Module Name*: «moduleName» extracted from «moduleName»_BswMCT.jar
- *Version*: stays empty
- *Execute*: Yes
- *Executable Path*: the path of the MCT relative to the tool executable and the name of the MCT: «moduleName»_BswMCT.jar
- *Output Path*: the path of the MCT relative to the tool executable

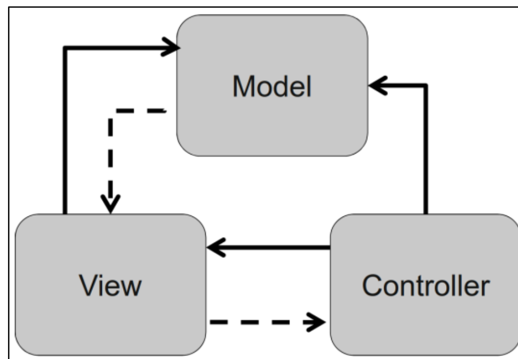

Figure 4.11. MVC Pattern

The implementation of the graphical user interface is solved by the Model View Controller (MVC) pattern. This design pattern is used to separate the logic from the representation of the data. The separation makes easier the later maintenance. Figure 4.11 shows the three components of the MVC pattern. The model contains the data, in this case, it would be the configuration of the tool. The view is responsible for displaying the data and what the user sees. The controller listens to the actions of the user. The dashed lines

correspond to communication via observers and observables. With this mechanism, a class (observer) monitors another class (observable). If there is a change the observer gets a notification.

The Implementation of MVC using UML classes is shown in Figure 4.12. The MainControl is the interface between the model, the ToolConfig and the user interface, the MainViewFrame. The controller handles all information and updates the data. Inputs of the user are recognized by action and focus listener and are processed respectively. The class diagrams of the classes for the BSG tool are shown in the Appendix C. The further implementation details of the BSG tool are not mentioned as it out of the scope of this thesis.

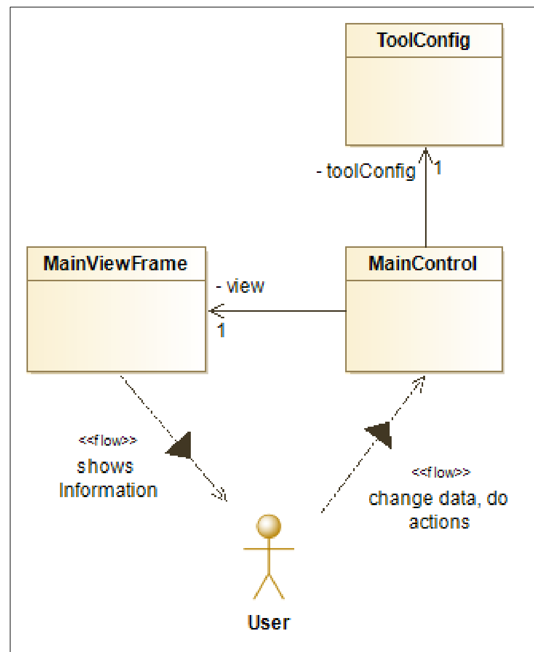

Figure 4.12. Class diagram for implemented MVC pattern

The functionality of the BSG tool shown is Figure 4.12 is explained. All the MCTs and are found and listed when the tool is opened. For the search, the Root Path is used as the root directory. All found templates are displayed on the left side. Each module in this

list has a checkbox next to its name to inform the user if a module is selected for generating or not. This option can also be changed on the right side at the item Execute. There are the two buttons Select All and Deselect All for a quick selection and deselection of all modules. The ECUC field contains the path and name of the ECUC Description file. If a module is selected, the fields of the lower right part are filled with the corresponding information, also called module properties. The Module Name and the Executable Path cannot be changed. The Version must be entered manually by the user, this is compared with the version which is stored in the template, if these do not match, the generation of that module will not be started and an error message is returned, therefore look at Table 4.7. This should

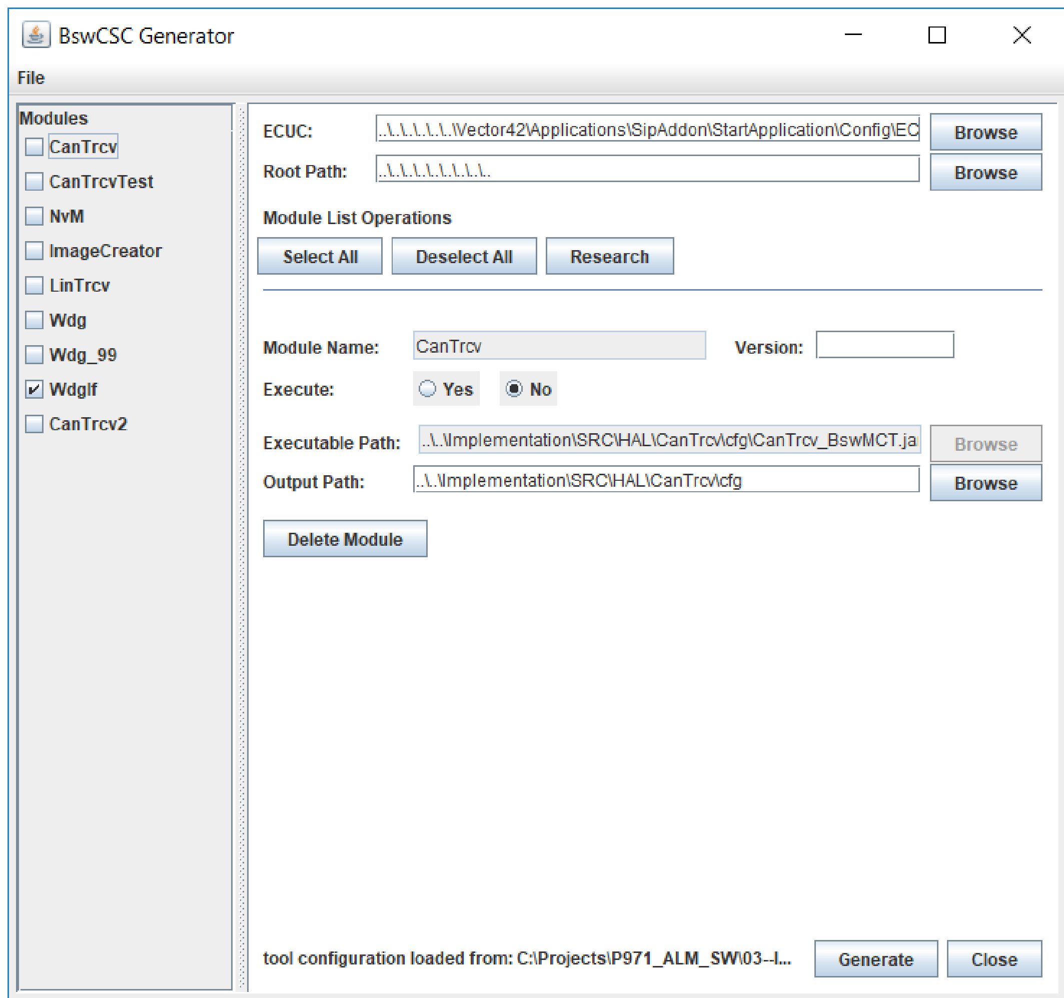

Figure 4.13. BSG tool with loaded config files

ensure that no wrong templates are used in the project. The button *Delete Module* deletes the currently selected module, which module properties are displayed on the right side. *Generate* triggers the generation of all selected modules by executing the templates. Before that, the tool configuration will be saved automatically. The *Close* button exits the program. Before the program is closed, it asks if the tool configuration should be saved, this is the difference to exiting the program via the X button, here the configuration is not saved. In the ribbon there is the item *File*, this opens a menu with 4 menu items:

- *Load tool config...:* opens the browser to select another tool configuration and loads it.
- *Save tool config*: saves the tool configuration.
- *Save tool config as...:* opens the browser to select a new path and name for the current tool configuration and stores it there.
- *Exit*: opens a dialog asking if the tool configuration should be saved, same as Close.

The tool can optionally be executed with command-line arguments. These arguments are the path and name of the tool configuration (-c) and a flag to enable the execution without GUI (-execute). An execution without arguments opens the GUI and loads the default tool configuration “BSGToolConfig.xml”. The program flow is shown graphically in Figure 4.14.

The program flow of BSG starts with the execution of the tool. The tool is saved within the development repository at APAG from where it can be executed. Once the module which needs to be generated is selected. The BSG interprets the arguments set for the module. If the arguments are found the tool configuration path is executed, if not the BSG accesses arguments from inside the tool executable. After the tool has the arguments the tool configuration is searched and loaded into the BSG. If the tool configuration is not found then an error message is displayed to inform the user that the configuration is not

found. If the configuration is loaded correctly, the user can generate the output source code using the generate button on the GUI.

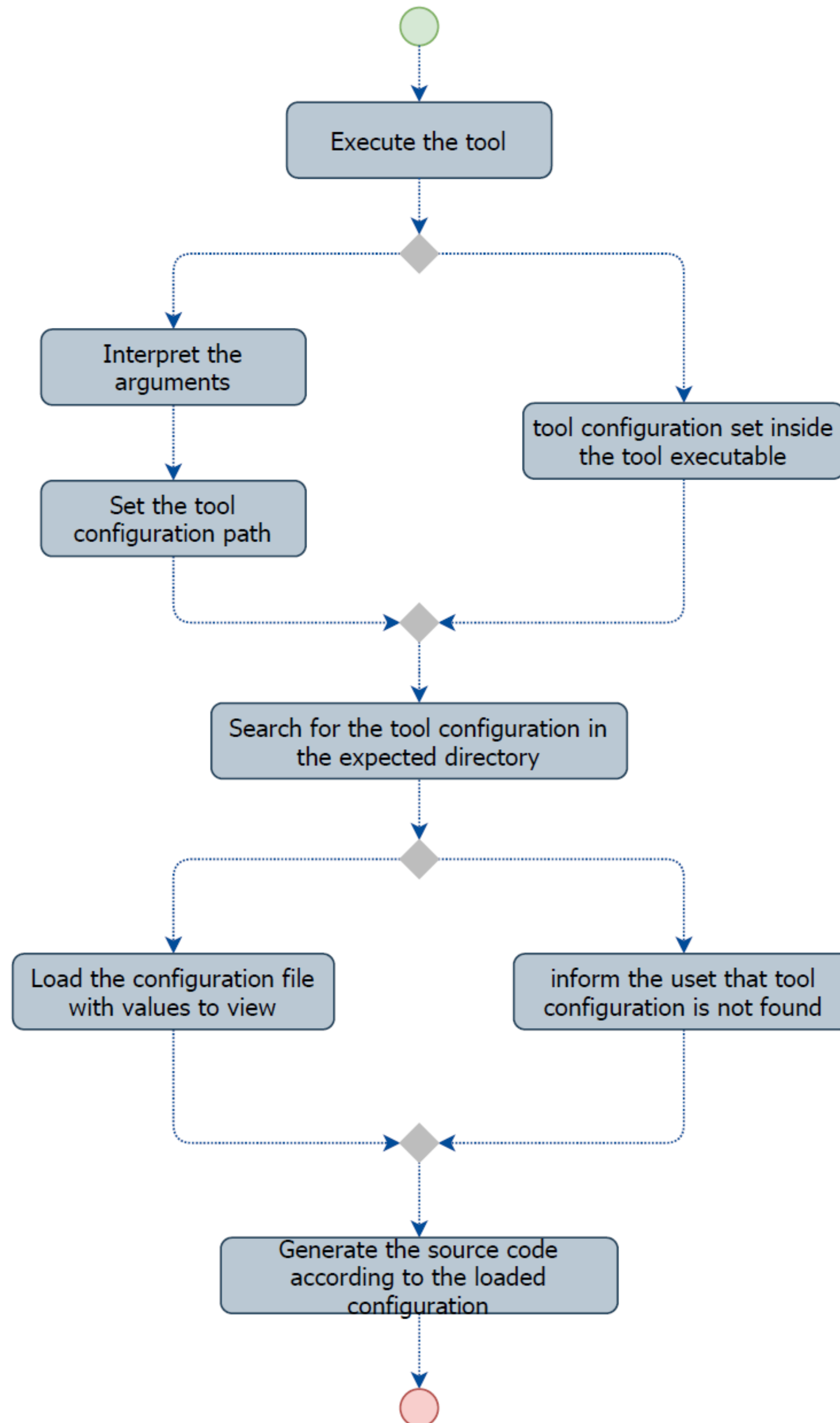

Figure 4.14. BSG tool flow

There is a feedback, shown in the GUI or command line, it is a simple line explaining if the generation was successful or failed. The detailed feedback is generated in the log file saved in the same folder as the BSG tool in APAG's file repository. The log file gives a detailed explanation for every step of tool execution. The feedback that can be received in the log file is explained in Table 4.7.

Table 4.7. Feedback in the log file

Execution Path of the tool: C:\TestProject\Tools\BSW_CSC_Generator

> *It prints the location the tool is executed. As will be shown later, it can happen that the path is not always as expected.*

Searching tool configuration in ".\"

> *This shows the location where the tool configuration is searched*

No argument for tool configuration is passed, as default BswCSCToolConfig.xml is used.

> *There is no argument transferred with the -c option, so the default one is used.*

Resolved absolute path of tool configuration: C:\TestProject \Tools \BSW_CSC_Generator\BswCSCToolConfig.xml\

> *This message shows the resolved absolute path, where the tool configuration is expected. This can be used to find potential errors if somethin went wrong by resolving the relative path.*

Tool configuration is not found.

> *There is no tool configuration, check the previous message. Is the path correct? It is no problem if it is the initial call of the tool or no tool configuration is expected.*

Searching templates recursively, starting with root folder C: \TestProject

> *The user searched for the modules, the root path which is used as starting point for the search is printed.*

Searching is finished, modules found.

> *The search for modules s finished. Some new modules are found. They are added to the modules list. At this point they won't appear in the tool configuration file because it hasn't been saved.*

Searching is finished, no new modules found.

> *The search for modules is finished. There are no new modules found. If a new module is expected check the root path and the location of the MCT.*

Tool configuration is saved in path C:\TestProject \Tools \BSW_CSC_Generator\BswCSCToolConfig.xml.

> *The tool configuration is saved in the path with the name "BswCSCToolConfig.xml"*

Tool configuration could not be saved in path: C:\TestProject \Tools \BSW_CSC_Generator\BswCSCToolConfig.xml. **Check settings and permissions!**

> *During the saving of the tool configuration an error occurred. Maybe it is because of missing permissions for the selected path.*

ECUC path is in tool configuration empty!
See: C:\TestProject \Tools \BSW_CSC_Generator\BswCSCToolConfig.xml

> *By triggering the generation, the ECUC path is checked. If it is empty this error appears and the generation will not be executed as it doesn't make sense to start it without the EcuCDesc.*

Start generating modules

> *This message shows that the generation process is started.*

EXECUTE:
java -jar
C:\TestProject\Implementation\SRC\HAL\CanTrcv\cfg\CanTrcv_BswMCT.jar
-ecuc C:\TestProject\EcuCExample.arxml **-o** C:\TestProject \Implementation\SRC\HAL\CanTrcv\cfg **-v** null

Output:
Errors and Warnings:

> *The command for the execution is printed, this can be used for debugging errors. In this example the -v argument is not set, so the template will return an error message and aborts the generation.*
> *The messages of the MCT are caught and printed in the "Output", "Errors and Warnings" sections.*

Error: Unable to access jarfile
C:\TestProject\Implementation\SRC\HAL\CanTrcv\cfg\CanTrcv_BswMCT.jar

> *This message shows that the MCT is not located at this path. If the template is moved to another location. Use the Delete button to remove the module and research to get the correct location. Don't forget to add the correct Version.*

Chapter 5. Functional Testing and Evaluation

This chapter discusses the results of the automated generation of ECUC. It discusses the generated code samples, test cases for error management of BSG and timing and cost analysis of our research.

5.1. Auto-Generate Source Code

After the BSG tool generates the code. The source code for the Wdg Module is generated and saved in the Wdg folder in the file repository. According to the requirements three source code files are generated for Wdg depending on the configuration classes: Wdg_Cfg.h, Wdg_Lcfg.c and Wdg_Lcfg.h. The complete generated source code cannot be discussed due to confidentiality reasons but a small part of the generated code corresponding to the xtend template shown in Figure 5.1 is shown in Figure 4.6.

```
const Wdg_ConfigType wdg_initialConfiguration_s =
{
        Wdg_SetWindowOpenTimeType:          WINDOW_OPEN_PERIOD_100_PERCENT,
        Wdg_SetErrorModeType:               ERROR_RESET_MODE,
        Wdg_SetIntRequestType:              INT_REQUEST_AT_75_PERCENT_DISABLED,
        Wdg_SetOverflowIntervalTimeType:    OVERFLOW_TIME_2HIGH9_DIVIDED_BY_WDTA_CLK,
};
```

Figure 5.1. Generate source code Wdg_Lcfg.c

The generated code printed everything that was coded in blue in Figure 4.6 as it is. We can see that the value for each of the parameters in the container WdgGeneral is accessed from the ECUC model and displayed in blue in Figure 5.1. These values were defined in the ARXML file and were accessed by xtend. Then the source code was auto-generated using the BSG tool.

A more complex example of the generated source code for the template shown in Figure 4.7 is given in Figure 5.2. The parameters were generated 4 times for the container SEIDType because of 4 different watchdog timers used in the hardware.

```
const WdgM_SEIDType WdgM_SEID_as[WDGM_NUM_OF_SEIDS] =
{
   {
      /* .WdgM_SEIDId_ui8 = */        WDGM_REF_ID_LIN1,
      /* .WdgM_SEIDUsed_b = */        TRUE,
      /* .WdgM_InitState_e = */       WDGM_TRCV_MODE_NORMAL,
      /* .WdgM_Start_b = */           TRUE,
      /* .WdgM_CheckpointRef_ui8 = */   0,
      /* .WdgM_TransitionRef_ui8 = */        0,
      /* .WdgM_Access_s = */            &WdgM_Access_SEIDLin1
   },
   {
      /* .WdgM_SEIDId_ui8 = */        WDGM_REF_ID_LIN2,
      /* .WdgM_SEIDUsed_b = */        TRUE,
      /* .WdgM_InitState_e = */          WDGM_TRCV_MODE_NORMAL,
      /* .WdgM_Start_b = */           TRUE,
      /* .WdgM_CheckpointRef_ui8 = */   0,
      /* .WdgM_TransitionRef_ui8 = */   0,
      /* .WdgM_Access_s = */            &WdgM_Access_SEIDLin2
   },
   {
     /* .WdgM_SEIDId_ui8 = */        WDGM_REF_ID_LIN3,
     /* .WdgM_SEIDUsed_b = */        TRUE,
     /* .WdgM_InitState_e = */          WDGM_TRCV_MODE_NORMAL,
     /* .WdgM_Start_b = */           FALSE,
     /* .WdgM_CheckpointRef_ui8 = */   0,
     /* .WdgM_TransitionRef_ui8 = */   0,
     /* .WdgM_Access_s = */            &WdgM_Access_SEIDLin3
   },
   {
     /* .WdgM_SEIDId_ui8 = */        WDGM_REF_ID_LIN4,
     /* .WdgM_SEIDUsed_b = */        TRUE,
     /* .WdgM_InitState_e = */          WDGM_TRCV_MODE_NORMAL,
     /* .WdgM_Start_b = */           FALSE,
     /* .WdgM_CheckpointRef_ui8 = */   0,
     /* .WdgM_TransitionRef_ui8 = */   0,
     /* .WdgM_Access_s = */            &WdgM_Access_SEIDLin4
   }
};

const WdgM_ConfigType WdgM_Config_s =
{
   /* .WdgM_SEID_ps = */ WdgM_SEID_as
};
```

Figure 5.2. Generated source code for WdgM_Cfg.c

5.2. Tests Cases for the BSG tool

The Table 5.1 defines test cases (TCs) that were created to check the functionality of the BSG tool. These briefly describe which action is taken and which result is expected. The test is passed if the actual response matches the expectation, if not it has failed. For some test cases, an execution on the command line is needed. The following convention is made to describe the location and command: Tilde (~) is used for the root path (here:

TestProject). The dollar symbol ($) is used to mark the beginning of the command. In addition, keywords in the command are highlighted in pink.

```
~/this/is/a/path/ $ this is a command
```

Figure 5.3. example command line

Table 5.1. Take Cases

Test Case No.	Description	Result
1. Starting tool		
1. 1 . Over icon (no arguments, so GUI is displayed)		
1. 1 .1	*Description:* Starting the tool without an existing default tool configuration. *Expected result:* User gets information that no tool configuration is found. All fields of the GUI are empty.	passed
1. 1 .2	*Description:* Starting the tool with existing default tool configuration. It is located in the same folder as the executable. *Expected result:* Loading the content of the *BswCSCToolConfig.xml*. Displays the content of the configuration. User get informed of the location of the loaded tool configuration.	passed
1. 2 . Over command line		
1. 2 .1	*Description:* Starting the tool over the command line without any arguments. *Expected result:* The reaction is similar to TC1.1.1 resp. TC1.1.2. *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator .jar`	passed
1. 2 .2	*Description:* Starting the tool over the command line with an other tool configuration. It is located in an other directory or has a different name compared to the default one. *Expected result:* The tool loads the content of the file and displays the content of the configuration. *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator.jar -c toolConfig/ValidToolConfiguration.xml`	passed

1. 2 .3	*Description:* Starting the tool over the command line with an invalid tool configuration (e.g. not found) *Expected result:* User gets information from the shell and GUI that no tool configuration is found. All fields of the GUI are empty. This case is comparable to TC1.1.1 *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator.jar -c toolConfig/InvalidToolConfiguration.xml`	passed
1. 2 .4	*Description:* The tool should be executed without opening the GUI. The tool configuration exists. *Expected result:* The tool loads the content of the *BswCSCToolConfig.xml* and starts the generation of the modules immediately without opening the GUI. The shell contains the information whether a module is generated or not. *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator.jar -execute`	passed
1. 2 .5	*Description:* The generation should be started with a valid configuration from a different location. The GUI should not be displayed. *Expected result:* The tool loads the content of the file and starts the generation of the modules immediately without opening the GUI. The shell contains the information whether a module is generated or not. *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator.jar -c toolConfig/ValidToolConfiguration.xml -execute`	passed
1. 2 .6	*Description:* The tool is executed with a invalid path resp. file name for the tool configuration. The GUI should not be displayed. *Expected result:* User gets information from the shell that no tool configuration is found and generation is aborted because of an empty ECUC path. *Command:* `~/Tools/BSW_CSC_Generator/ $ java -jar BswCSC_Generator.jar -c toolConfig/InvalidToolConfiguration.xml -execute`	passed

1. 2 .7	*Description:* The tool is called by the shell from an other location. *Expected result:* The tool loads the content of the default tool configuration which is located in the same path were the tool executable is located and displays the content of the configuration. This case is comparable to TC1.1.2 *Command:* `~/Tools/ $ java -jar BSW_CSC_Generator/BswCSC_Generator.jar`	passed
2. Loading configuration		
2. 1 . automatically at start up		
2. 1 .1	*Description:* The tool configuration is found in the expected folder and is loaded at start up. *Expected result:* The configuration is displayed in the graphical user interface. The modules are listed, the check boxes show the selection for the generation. The first module's configuration in the list is displayed in more detail.	passed
2. 1 .2	*Description:* The tool configuration is not found in the expected folder, so no tool configuration is loaded. *Expected result:* The view stays empty. At the bottom an info shows "No tool configuration found".	passed
2. 2 . via menu		
2. 2 .1	*Description:* (Re-)Loading the tool configuration via the menu item *Expected result:* The tool shows the new configuration in the graphical user interface. The modules are listed, the check boxes show the selection for the generation. The first module's configuration in the list is displayed in more detail.	passed
3. Saving configuration		
3. 1 . via menu		
3. 1 .1	*Description:* The tool configuration will be saved by using the menu (File -> Save). *Expected result:* The current configuration is saved. The loaded tool configuration is over written with the changed values. The GUI shows a massage at the bottom, that saving was successfully or has failed.	passed

No.	Test case	Result
3. 1 .2	*Description:* The tool configuration will be saved by using the menu (File -> Save as..) *Expected result:* The tool opens the file explorer to select a file. Saving the current configuration in the selected file. The GUI shows a massage at the bottom, that saving was successfully or has failed. The new file will be used for further runtime.	passed
3. 2 .	**by generating**	
3. 2 .1	*Description:* By clicking the generate button the configuration will be automatically stored to maintain the coherence between generated source code and the configuration. *Expected result:* The tool overwrites the loaded configuration with the changed values.	passed
3. 3 .	**by closing**	
3. 3 .1	*Description:* If the tool will be closed through the button or over the menu with exit (compare with TC4), a dialog asks the user if the configuration should be stored. *Expected result:* Depending of the user's decision the tool configuration is saved or discarded	passed

No.	Test case	Result
4.	**Closing the tool**	
4. 1 .	**by Close button**	
4. 1 .1	*Description:* The tool will be closed through the close button. *Expected result:* A dialog opens, asking the user if the tool configuration should be saved or not. Afterwards the application is closed.	passed
4. 2 .	**by menu**	
4. 2 .1	*Description:* The tool will be closed via the menu (File -> Exit). *Expected result:* A dialog opens, asking the user if the tool configuration should be saved or not. Afterwards the application is closed.	passed
4. 3 .	**by the X**	
4. 3 .1	*Description:* The tool will be closed through the X in the upper right corner of the window *Expected result:* The application is closed immediately. The changed configuration is not saved.	passed

5. Elements of the view		
5. 1 . Browse button		
5. 1 .1	*Description:* There is no path entered in the left text field and the browse button to its right is clicked. *Expected result:* The file explorer is opened. The start directory is the location of the tool's executable.	passed
5. 1 .2	*Description:* There is a path entered in the left text field and the browse button to its right is clicked. *Expected result:* The file explorer is opened. The start directory is the path of the text filed.	passed
5. 1 .3	*Description:* The browse button for searching an *EcuC Description* is pressed. *Expected result:* The file explorer only shows folders and files with the pattern *.arxml.	passed
5. 1 .4	*Description:* The browse button for selecting the root path is pressed. *Expected result:* The file explorer only shows folders.	passed
5. 1 .5	*Description:* The browse button for selecting the output path is pressed. *Expected result:* The file explorer only shows folders.	passed
5. 2 . Browse text filed		
5. 2 .1	*Description:* A absolute path is entered in the text field. *Expected result:* The entered path is converted to a relative path to the location of the executable.	passed
5. 2 .2	*Description:* A relative path is entered in the text field. *Expected result:* The entered path should be relative to the to the location of the tool's executable elsewise an error will occur by resolving the path.	passed
5. 3 . Version text filed		
5. 3 .1	*Description:* A version string is inserted. There are no rules defined how the format should look like. (see also TC 5.10.2) *Expected result:* The entered text is displayed.	passed

5. 4 . Modules list		
5. 4 .1	*Description:* The settings of a module are regarded by clicking on its name in the module list. *Expected result:* The module's configuration is shown on the right side. The sate for execution has changed.	passed
5. 5 . Select All button		
5. 5 .1	*Description:* All modules should be selected for the generation, therefore the select all button is used. *Expected result:* All checkboxes appears as selected in the modules list. The first module in the list is displayed.	passed
5. 6 . Deselect All button		
5. 6 .1	*Description:* All modules should be deselected for the generation, therefore the deselect all button is used. *Expected result:* In the modules list none checkbox is selected. The first module in the list is displayed.	passed
5. 8 . Research button		
5. 8 .1	*Description:* The button to research MCTs in the project is pressed. *Expected result:* The tool searches through all subfolders of the root path and lists founded MCTs.	passed
5. 9 . Delete button		
5. 9 .1	*Description:* The delete button is pressed to remove one module. *Expected result:* The selected module is removed from the modules list. User get informed which module was deleted. The first module in the list is displayed.	passed
5. 9 .2	*Description:* The delete button is pressed but no module is selected (e.g. no modules are in the list). *Expected result:* Nothing happens.	passed

5.10. **Generate button**		
5.10.1	*Description:* The generate button is pressed to start the generation. Everything is configured correctly. *Expected result:* The module configurations are generated to the respective output path. The user get informed that the generation is finished successfully.	passed
5.10.2	*Description:* The generate button is pressed to start the generation. Something is wrong (e.g. Version is not correct or template has a problem). *Expected result:* The user get informed that generation is finished, but that there was an error. He has to look in the log file (*BswCSCLogFile.log*) for further information.	passed
5.10.3	*Description:* The generate button is pressed to start the generation. The ECUC path is empty. *Expected result:* The user get informed that generation is canceled, because of the empty ECUC path.	passed
5.11. **Close button**		
5.11.1	*Description:* The close button is pushed for closing the tool. *Expected result:* See TC4.1.1	passed
5.12. **File menu**		
5.12.1	*Description:* The file item in the menu ribbon is pressed. *Expected result:* Further options for saving loading and exiting the tool are listed.	passed

All the test cases were put to test at APAG and all were verified and passed. This proves that the BSG tool works without any issues

5.3. *Approximate Time and Cost comparisons*

Time and Cost for the automated generation of ECUC in AUTOSAR utilized for this research is compared with the generation of ECUC manually done by the developers. Figure 5.4 gives an approximate estimation of the time taken for auto-generating ECUC configurations at APAG versus the time taken to generate them manually by a developer. The work done manually by a developer to generate the ECUC is approximately around 8 weeks. But our research at APAG can auto-generate the configurations and complete the whole process in 2 weeks.

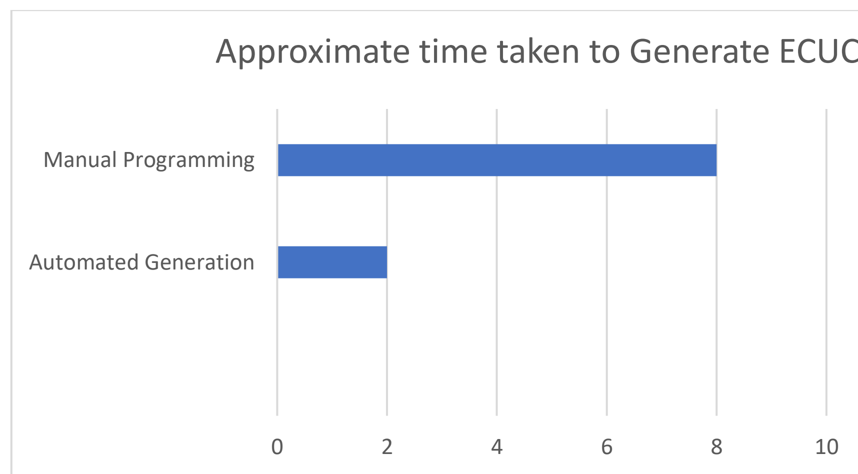

Figure 5.4. Approximate time taken to generate ECUC

Automotive suppliers use some existing tools that our in the market like Vector da Vinci [27] to complete the ECUC process. These tools are very expensive for small companies like APAG Elektronik. It is easier for small companies to develop their own code generator tool that will turn out to be less expensive.

Chapter 6. Conclusion

6.1. Summary

In this research, we described the process for the automated generation of the ECU configurations, which is a necessary process for developing an ECU based on AUTOSAR. The MCT and the BSG generator can be used in various projects to generate the configuration files automatically with minimal input and effort from the developer. The aim was to develop a process in the automotive embedded systems to ensure consistency in design, software implementations, driver configurations and reduce the time and cost consumed by the AUTOSAR ECU generation process. This work presents an approach that seamlessly describes the ECU configuration process using templates and a generation tool that can be reused. This approach can be used for all the AUTOSAR modules present in the basic software layer.

The whole ECU configuration process and source code generation without the BSG CAD tool can be done manually by a group of 2 to 3 embedded developers over a period of approximately one month for one module of AUTOSAR. But by using the BSG tool and the MCTs we can finish the ECU configuration process and generation of one module in approximately one week.

Improved efficiency, traceability, and consistency for the configuration process, reduction in cost, time and cumbersome, error-prone manual work along the ECU development path are the main benefits of this approach. The approach can be further improved in the process of configuring of the RTE in AUTOSAR and an even faster implementation process.

6.2. Future Work

This research has a future scope of improvement. This process can be implemented with the new AUTOSAR adaptive platform. Sorting algorithms can be used for the module search to make the code generation faster. The template descriptions can be made simpler by using domain-specific language. These optimizations and improvements can be implemented through further research.

References

[1]. Guido Sandmann and Richard Thompson. "Development of AUTOSAR Component with Mode-Based Design," 2008, doi:10.4271/2008-01-0383.

[2] Georg Macher, Eric Armengaud and Christian Kreiner "Automated Generation of AUTOSAR Description File for Safety-Critical Software Architectures," presented at Informatik – Automotive Software Engineering Workshop, Stuttgart, 2014.

[3]. Fabrizio Fabbrini, Maario Fusani, Giuseppe Lami, et al., "Software Engineering in the European Automotive Industry: Achievements and Challenges," published in 32nd Annual IEEE International Computer Software and Applications Conference, Finland, 2008, doi: 10.1109/COMPSAC.2008.140.

[4]. Georg Macher, Rene Obendrauf, Eric Armengaud, et al., "RTE Generation and BSW Configuration Tool-Extension for Embedded Automotive Systems," presented at European Congress Embedded Real Time Software and Systems, 2016.

[5]. AUTOSAR, "AUTOSAR XML Schema," AUTOSAR standard 4.4, August. 2018.

[6]. JaxEnter, "Xtend Programming language," https://jaxenter.com/xtend-pirates-jvm-efftinge-132385.html, accessed Aug 2019.

[7]. Eclipse, "Eclipse IDE – Open platform for professional developers," https://www.eclipse.org/eclipseide/, accessed Fed 2018.

[8]. AUTOSAR, "AUTOSAR introduction," https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_Introduction.pdf, accessed Nov. 2019.

[9]. AUTOSAR, "AUTOSAR Layered Architecture," AUTOSAR standard 4.4, August. 2018.

[10]. AUTOSAR, "Requirements on RTE Software," AUTOSAR standard 4.4, August. 2018.

[11]. AUTOSAR, "Specification of Watchdog Driver," AUTOSAR standard 4.4, August. 2018.

[12]. AUTOSAR, "Specification of Watchdog Interface," AUTOSAR standard 4.4, August. 2018.

[13]. AUTOSAR, "Specification of Watchdog Manager," AUTOSAR standard 4.4, August. 2018.

[14]. AUTOSAR, "Technical Overview," AUTOSAR standard 4.4, August. 2018.

[15]. Kunal Chandmare, "Automated Configuration of Time-Critical Multi-Configuration AUTOSAR Systems," TU Chemnitz, 2017

[16]. AUTOSAR, "AUTOSAR Methodology," AUTOSAR standard 1.2.2 Rev 3.2, April. 2007.

[17]. AUTOSAR, “AUTOSAR Methodology,” AUTOSAR standard 4 Rev 4.4, August. 2018.

[18]. Miroslaw Staron. Automotive Software Architectures-An Introduction. Springer, 2017. isbn: 978-3-319-58609-0.

[19]. AUTOSAR, “AUTOSAR ARXML Serialization Rules,” AUTOSAR standard 4 Rev 4.4, August. 2018.

[20]. Xtend, “Xtend Documentation,” https://eclipse.org/xtend/documentation/index.html, accessed November, 2018.

[21]. AUTOSAR, “AUTOSAR,” https://www.autosar.org/, accessed March, 2019.

[22]. AUTOSAR, “Specification of ECU Configuration,” AUTOSAR standard 4.4, August. 2018.

[23]. Guido Sandmann and Richard Thompson. “Development of AUTOSAR Component with Mode-Based Design,” 2008, doi:10.4271/2008-01-0383.

[24]. Brett Murphy, Chris Hayhurst, Jon Friedman, et al., “Verification and Validation Integration within Processes Using Model-Based Design,” 2008, doi:10.427/2008-01-2709.

[25]. J.-C. Lee and T.-M. Han.” ECU Configuration Framework based on AUTOSAR ECU Configuration Metamodel,” Presented at International Conference on Hybrid Information Technology 2009. 2008, doi:10.1145/1644993.164043.

[26]. H. C. Jo, S. Piao, and W. Y. Jung “Design of a Vehicular code generator for Distributed Automotive Systems,” presented at Seventh International Conference on Information Technology 2010, USA, 12-14 April. 2010, doi: 10.1109/ITNG.2010.212.

[27]. Stefan Voget and Continental Engineering Services GmbH “AUTOSAR and the Automotive Tool Chain,” presented at Design, Automation and Test Conference & Exhibition 2010, Europe, 8-12 March. 2010, doi: 10.1109/2010.5457202.

[28]. Georg Macher, Rene Obendrauf, Eric Armengaud, et al., “RTE Generation and BSW Configuration Tool-Extension for Embedded Automotive Systems,” presented at European Congress Embedded Real Time Software and Systems, 2016.

[29]. AUTOSAR, “Requirements of ECU Configurations,” AUTOSAR standard 4 Rev 4.3.1, August. 2017.

Appendix A. AUTOSAR Methodology 4.4

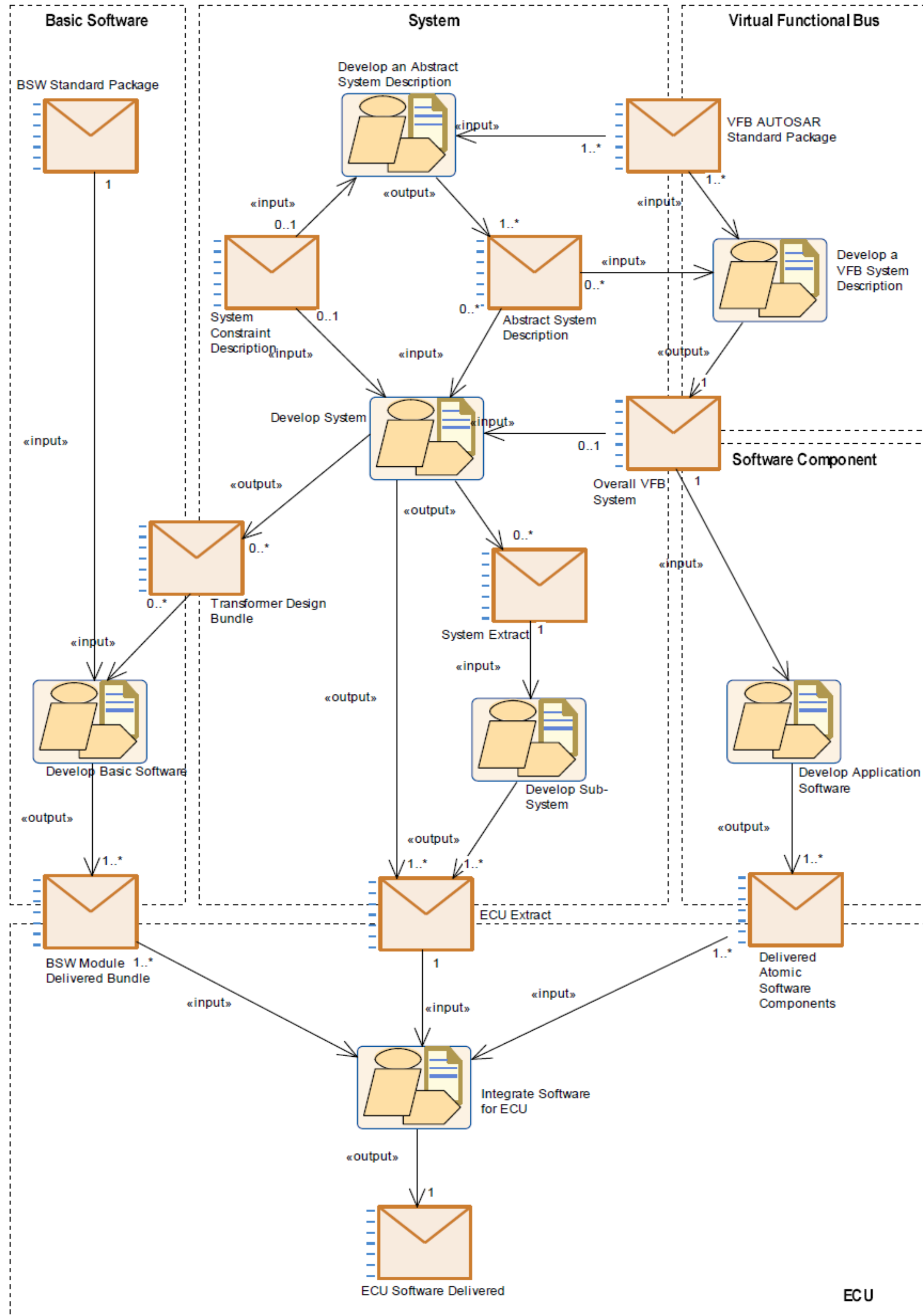

Figure A.1. AUTOSAR Methodology (version 4.4 [16])

Appendix B. Executable Java Code

```
package com.apagcosyst.genWdg;

import java.io.BufferedWriter;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import com.apagcosyst.generator.ArgumentsInterpreter;
import com.apagcosyst.generator.IGenSourceCode;
import com.apagcosyst.ecuCModel.ArxmlFileInterpreterClass;
import com.apagcosyst.ecuCModel.EcuConfigurationModel;
import com.apagcosyst.ecuCModel.ModuleConfigurationClass;

public class Wdg_Main {
	private String moduleName;
	private String moduleVersion;
	private ModuleConfigurationClass moduleCfg;
	private EcuConfigurationModel ecuConfig;
	private IGenSourceCode moduleGen;
	private FileWriter fwLcfgc;
	private FileWriter fwLcfgh;

	public Wdg_Main()
	{
		moduleName = "Wdg";

		/* this is filled in Setup() function */
		moduleCfg = new ModuleConfigurationClass();
		ecuConfig = new EcuConfigurationModel();
		moduleGen = new Wdg();
		moduleVersion = new String("1.0”);
	}
	public static void main(String[] args)
	{

		ArgumentsInterpreter arguments = new ArgumentsInterpreter();
		arguments.interpret(args);

		if(arguments.isValid())
		{
			Wdg_Main module = new Wdg_Main(/*arguments.getVersion()*/);

			if(arguments.getVersion().equals(module.moduleVersion))
			{
			try {
			      module.ecuConfig.loadArxmlFileToModel(arguments.getEcucPath());
			module.moduleCfg = module.ecuConfig.getModuleConfigurationByName(module.moduleName);

			if(module.moduleCfg.getShortName().equals(ArxmlFileInterpreterClass.invalid))
			{
			  System.err.println("ModuleConfiguration: " + module.moduleName + " in EcuC File: " +
			arguments.getEcucPath() + " not found!\n!!Generation aborted!!!");
			  return;
			}
			else
			{
			  String output = arguments.getOutputRootPath() + File.separator + module.moduleCfg.
			  getShortName();
```

```
                    module.fwCfgc = new FileWriter(output + "_Cfg.c");
                    module.fwLcfgc = new FileWriter(output + "_Lcfg.c");
                    module.fwLcfgh = new FileWriter(output + "_Lcfg.h");

                }

            } catch (Exception e) {
                e.printStackTrace();
            }

            try {
                /* Generate the «module»_Cfg.c */
                BufferedWriter bwCfgc = new BufferedWriter(module.fwCfgc);
                bwCfgc.write((String) module.moduleGen.GenCfgC(module.moduleCfg,
                module.ecuConfig));
                bwCfgc.close();
                module.fwCfgc.close();

                /* Generate the «module»_Lcfg.c */
                BufferedWriter bwLcfgc = new BufferedWriter(module.fwLcfgc);
                bwLcfgc.write(module.moduleGen.GenLcfgC(module.moduleCfg,
                module.ecuConfig));
                bwLcfgc.close();
                module.fwLcfgc.close();

                /* Generate the «module»_Lcfg.h */
                BufferedWriter bwLcfgh = new BufferedWriter(module.fwLcfgh);
                bwLcfgh.write(module.moduleGen.GenLcfgH(module.moduleCfg,
                module.ecuConfig));
                bwLcfgh.close();
                module.fwLcfgh.close();

                System.out.println("Generation is finished!");

            } catch (IOException e) {
                // Auto-generated catch block
                e.printStackTrace();
            }
        }
        else
        {
            System.err.println("Version is not valid!" + System.getProperty("line.separator") +
            "Expected: " + module.moduleVersion + System.getProperty("line.separator") +
            "got: -v " + arguments.getVersion() + System.getProperty("line.separator"));
        }
    }
    else
    {
        System.err.println("Arguments are not valid" + System.getProperty("line.separator") +
        "Expected: -v [Version] -o [outputPath] - ecuc [EcucLocation]" + System.getProperty
        ("line.separator")  + "got: -v " + arguments.getVersion() + System.getProperty("line.separator")
    }

  }

}
```

Appendix C. BSG class diagram

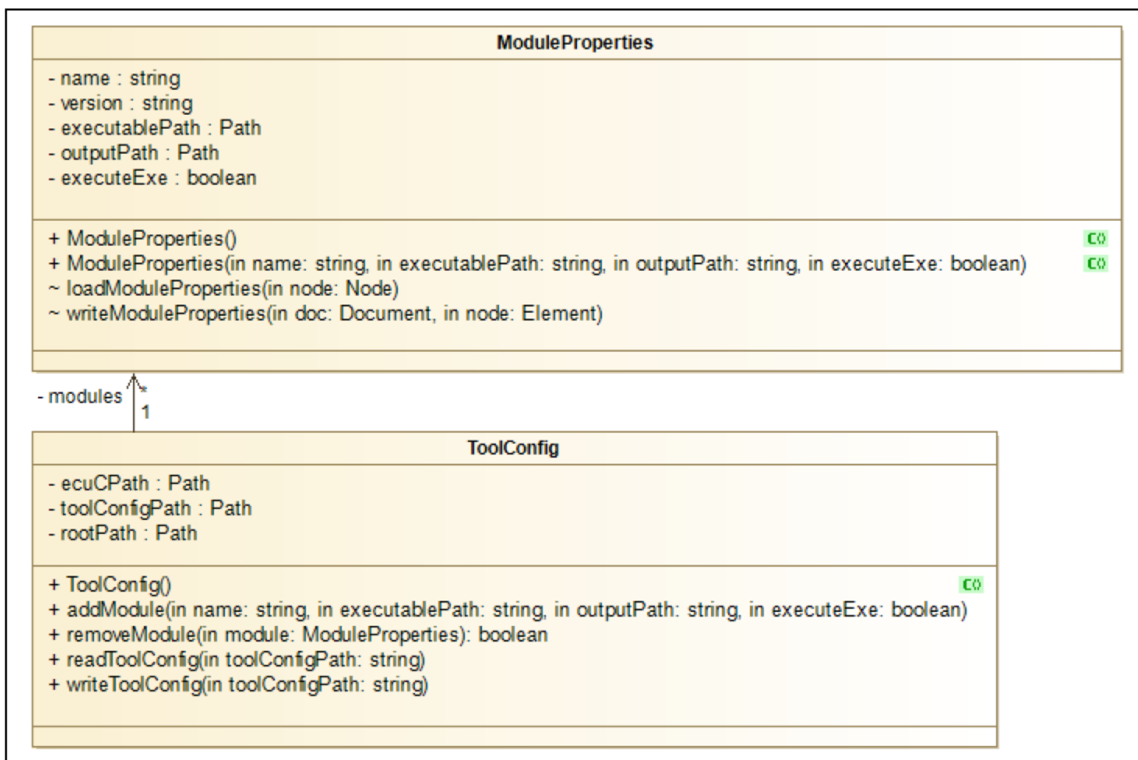

Figure A.2. Class diagram for the BSG

Vita Auctoris

NAME:	Usha Sreeram
PLACE OF BIRTH:	Bengaluru, India
YEAR OF BIRTH:	1995
EDUCATION:	Bachelor of Engineering in Electronics and Communication Engineering Dr. Ambedkar Institute of Technology, Bengaluru, India 2013 – 2017
	Master of Applied Science in Electrical and Computer Engineering University of Windsor, Windsor, Ontario, Canada 2017-2019