Efficient Test Case Generation for AUTOSAR Basic Software Code Generators

Master of Science Thesis
in the Computer Science and Software Engineering Programmes

Ger Garrigan, Daniel Ivan

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

Göteborg, Sweden, June 2013

Efficient Test Case Generation for AUTOSAR Basic Software Code Generators

Ger Garrigan, Daniel Ivan

Examiner: Christian Berger

Supervisor: Matthias Tichy

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

SE-412 96 Göteborg, Sweden

Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Göteborg, Sweden June 2013

Table of Contents

Figures

Abstract

In the contemporary automotive industry, the complexity of software architectures for electronic control units (ECUs) has increased drastically. Aiming to improve the complexity management of these architectures, a worldwide partnership of car manufacturers and suppliers has created a standardized approach called AUTOSAR (AUTomotive Open System ARchitecture) (AUTOSAR Basics, 2012). At the highest abstraction level, the architecture of AUTOSAR contains three software layers which run on a Microcontroller. These three layers are Application Layer, Runtime Environment (RTE) and Basic Software (BSW) (AUTOSAR Layered software architecture, 2011). The BSW layer is further divided into multiple software modules which provide basic services such as memory management and bus communication (Mecel, 2013). These software modules can be configured to satisfy the needs of the customer. Testing these configurations requires a large amount of effort and time, especially since they are manually generated. This thesis deals with the automatic generation of these test cases, the configurations of the BSW modules. Two test case generation approaches were developed and compared. The first is random generation where elements to be added to the test case are chosen in a random manner. The second is pairwise generation where elements are added to the test case based on satisfying all pairs of element values. The experiments conducted to compare the two generation techniques ran the configurations created for three BSW modules through their SCGs (Source Code Generators) and showed that both techniques have the ability to uncover problems within a SCG. This thesis was conducted as a case study at Mecel AB in Gothenburg.

Acknowledgements

We would like to thank our university supervisor Matthias Tichy for his dedication and enthusiasm throughout this thesis. We would also like to thank Mecel AB for providing a productive and friendly work environment, especially our supervisor Samuel Sveningsson for his guidance, Christian Lidberg and the other members of the Picea team for the help they gave us.

Ger would like to thank his parents for their unfailing encouragement and support.

Daniel would like to thank his parents for their support.

1. Introduction

Manual integration testing is fraught with issues. When considering the integration of different components and their parameters, the combinatorial complexity can become too large to make manual testing feasible. Edge cases can easily be missed. Manual testing relies heavily on humans, is prone to error and is a time intensive process. According to Claessen et al., test automation allows for faster completion of testing or using the time allocated for more thorough tests. Testing accounts for up to 50% of the cost of software development. (Claessen et al., 2000).

So it is time to automate the generation of integration tests cases. This is the area in which our thesis lives. This thesis was carried out in Mecel AB, a company who develops automotive software. They are members of AUTOSAR, a collaborative initiative that facilitates structured development of automotive software driven by a common schema, the AUTOSAR schema (Honekamp, 2009). Using this schema Mecel configure AUTOSAR Basic Software (BSW) modules. Each module has a Source Code Generator (SCG). The BSW configurations are then fed into their SCG to generate code to be run in the vehicles.

Our thesis is comprised of three main parts. First is research into test case generation techniques by analysis of literature in the areas of integration testing, test case generation and software product lines. Second is the development of some of the identified techniques with the aim of automating the creation of AUTOSAR configurations. These generated AUTOSAR configurations are the test cases for the SCG. Third is the comparison of the techniques through experimentation. The techniques to be compared are random generation and generation using the pairwise method which is based on combinatorial mathematics. The aim of the test case generation is to produce configurations likely to cause the SCG to crash and thus uncover unhandled issues within the SCG or test cases likely to identify bugs in the SCG.

1.1. AUTOSAR

"AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry." (AUTOSAR Home, 2012)

The increasing usage of software within vehicles results in more complex architectures and with the increase in requirements comes an increase in conflicting requirements. AUTOSAR provides a common platform for defining the architecture and software to meet the varied requirements. This common platform leads to a structured and clearly defined architecture (AUTOSAR Motivation and Goals, 2012). Unless otherwise stated the following description of AUTOSARs layered architecture is based on the reference (AUTOSAR Layered Software Architecture, 2011). The layered architecture defined by AUTOSAR can be seen in figure 1.

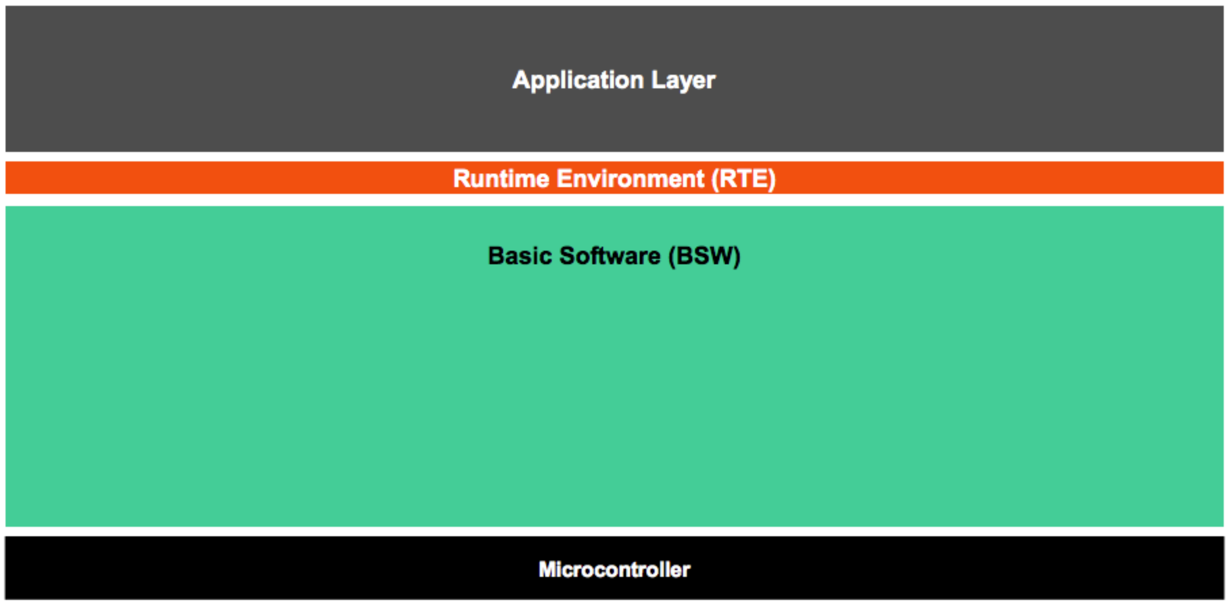


Figure 1 - High Level View of AUTOSAR Layered Architecture.

Figure 1 shows the four layers that make up the AUTOSAR architecture. The top layer, the **Application Layer** contains software components (SW-C). These SW-Cs contain software which runs in the AUTOSAR system. Software that controls the brightness of the headlamps according to external stimulus is an example of the type of software that is contained within the application layer (AUTOSAR Technical Overview, 2008). These software components are connected to each other via the Virtual Functional Bus (VFB). These connections are hardware independent. The **Runtime Environment (RTE)** realizes these connections thus providing concrete connections between components within the application layer and between the application and Basic Software layers (AUTOSAR VFB, 2011). The **Basic Software (BSW)** layer is composed of standard software modules for areas such as OS communication, memory, hardware drivers, diagnostics etc. The **Microcontroller** is the Electronic Control Unit (ECU) hardware layer on which the software is run.

1.1.1 Basic Software

The BSW is configured using parameter definition files. These files define the structure that the BSW configuration must conform to. The parameter definition files contain elements such as containers, parameters and references. For example the Development Error Tracer (Det) (AUTOSAR Det, 2011) module has a general mandatory container containing such parameters as DetForwardToDlt which indicates if the Dlt (Diagnostic Log and Trace) interface is required and when set to true, Det will forward its call to a function called Dlt_DetForwardErrorTrace. It also contains an optional notification container which when selected contains a parameter called DetErrorHook for defining calls to error hooks in code. References allow a parameter to reference a different container within the same module or within an external module. The BSW modules in AUTOSAR 4.0 are contained in figure 2.

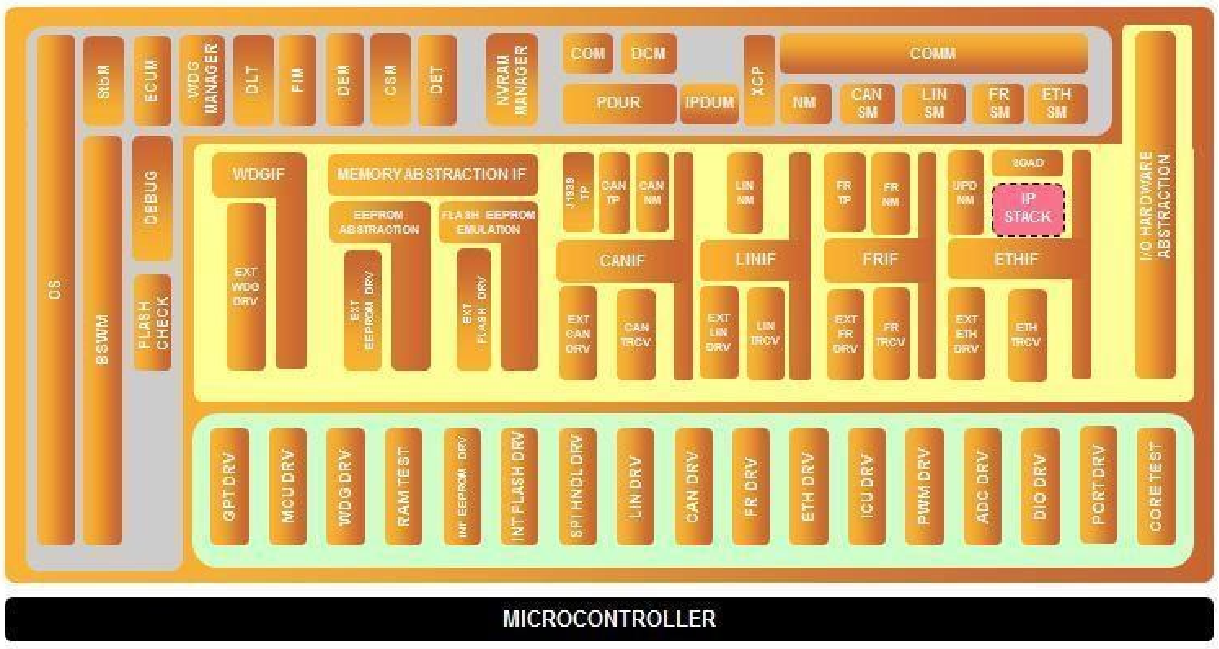


Figure 2 - Basic Software modules (4.0).

When the BSW configurations have been created they are run through a Source Code Generator (SCG) in order to generate C code. The focus of this thesis is to compare methods for automated creation of test configurations for the BSW. Before conducting this study all configurations that were processed by the SCG were created manually. The motivation behind automation is to test the SCG with configurations that may not have been considered previously. These configurations conform to the parameter definitions. The SCG should be able to handle any configuration conforming to the parameter definition without crashing.

1.2. The Generators

Two comparable approaches have been identified for generating the configurations, a random configuration generator and a pairwise configuration generator.

1.2.1. The Random Generator

The random generator takes in the parameter definition and randomly selects the multiplicities and values to be output to the configuration. The random generator is used as the baseline against which the pairwise generator can be compared.

Random test generation has the advantages that it is relatively cheap and fast to generate test cases. It can detect a large number of errors, is most likely to mimic real world scenarios and can be used to see how reliable a program is (Oriat, 2005). We see additional advantages, the first being that there is a greater likelihood of testing unusual configurations due to its random nature when compared with manually creating configurations. This is because manual configurations are usually created based on customer requirements whereas the random approach is not choosing the contents of the configurations based on any preconceptions. Second, the random generator can also generate many configurations for testing the SCG.

It has the disadvantage that a lot of the produced tests can be deemed irrelevant (Oriat, 2005). Following on from Oriat we see that the random generator is not aware of previously created configurations and so may produce duplicates. We also see that the number of possible configurations that can be created from a parameter definition is so large as to make their creation infeasible.

1.2.2. The Pairwise Generator

The use of software product lines involves identifying common features that can be used across an organization's product range. The use of a common base allows for increased quality and faster development time. When a product requires a feature that is specific to just that product a variability point is introduced. Each variation point increases combinatorial complexity and the adding of variation points is carefully controlled. Software product lines can be tested using a technique derived from combinatorial mathematics called pairwise (McGregor, 2010).

The parameter definitions for the BSWs contain mandatory and optional elements. For example the Det module introduced earlier has one mandatory container and one optional container. The parameter definitions can be considered as a self-contained software product line. As the pairwise technique is used for testing software product lines, this indicates that it can also be used for testing when working with parameter definitions.

The pairwise testing approach uses combinatorial theory to significantly reduce the number of configurations required that can still find 90% of the faults that exhaustive methods can find (McGregor, 2010). Pairwise considers that most faults occur with single values or between pairs of values and so reduces the configurations to just those that cover all pairs (Cohen et al., 1996). A definition containing 120 binary parameters requires 10 test cases (configurations in our case) using pairwise as opposed to 2^{120} using exhaustive testing (Cohen et al., 1996).

We see the advantages that fewer configurations are needed as test cases, which are still likely to produce a crash. Also, it does not produce duplicate configurations since each configuration covers at least one new pair. The pairwise approach is also deterministic, so the configurations will not change from one run to the next for the same input. This means that pairwise can be run once and its configurations reused many times.

We see a disadvantage in that pairwise does not have 100% fault coverage. Also, it is possible to use n-wise testing where n can be higher than 2. We used pairs due to the indications of its benefits from the literature referenced above.

1.3. Comparison of Generators

In order to compare the two generators, we considered three different criteria. The first was the number of crashes found based on a set number of configurations. So both the pairwise generator and the random generator created the same number of configurations and ran them through the SCG for the same module and the number of crashes produced was recorded. The reason for this comparison is that pairwise produces a set number of configurations so we were eager to see how the generators compare when the random generator was also given a set number of configurations.

The next comparison was to record the number of crashes identified when both generators are allowed to generate configurations and run them through the SCG for a set amount of time. The reason for this comparison is that the pairwise approach only requires a set number of configurations whereas random may prove beneficial if it can continue to generate as many configurations as it can and run them through the SCG in a given amount of time.

Finally we looked at the overhead of the time taken to generate the configurations introduced by pairwise when considering the time needed to run them through the SCG. It is interesting to see if there is an extra overhead for the pairwise generator to create the configurations when compared with the random generator and whether this overhead is noteworthy when considering the time it takes to run the configurations through the SCG.

1.4. Experiments

The scope of the experimentation was to analyse the process of taking parameter definitions as input to the generators and using the resulting configurations to try and find crashes or bugs in the SCG. The SCG has the ability to produce code but analysis of the code produced is outside the scope of this thesis. The reason for this is that of the unit tests that Mecel AB create there are currently none that exist for testing the code we produce.

The pairwise and random generators were run on a nightly builder. Three BSW modules were used in the experimentation. These were:

- Development Error Tracer (Det): A module with 7 elements (2 containers and 5 parameters) and low complexity. This module has low complexity as the only dependencies that need to be considered are parent-child relationships. The pairwise generator has a certain amount of overhead when creating its test cases. This overhead should be minimal with the Det module and the number of test cases created by pairwise should be small due to the small number of elements and low complexity.

- Function Inhibition Manager (FiM): A medium sized module with 35 elements (8 containers and sub-containers, 1 choice container with 2 choices, 15 parameters and 9 references). The FiM module has greater complexity than Det as it uses references, choices and has more levels in the parent-child hierarchy. As the complexity increases the pairwise generator could identify crashes or bugs more effectively than the random generator as it requires many less configurations. At this stage maybe the time taken to run the pairwise generator will have an effect on its ability to produce the configuration test cases within a suitable timeframe e.g. one run of a nightly builder.

- Diagnostic Event Manager (Dem): A large module with 195 elements (34 containers and sub-containers, 2 choice-containers with 3 choices each, 120 parameters and 33 references). The complexity is higher than FiM due to the larger number of references and choices. Here the time taken for pairwise to produce the configuration test cases could become more prominent. As random produces one configuration test case at a time it is interesting to see if pairwise can achieve its goals within one nightly build run.

The configuration test cases that were generated for these modules were run through their SCGs. The following metrics were recorded:

- The number of pairwise configurations created and run through the SCG.

- The number of random configurations created and run through the SCG.

- The number of times the SCG crashed using the pairwise configurations.

- The number of times the SCG crashed using the random configurations.

- The number of errors the SCG produced using the pairwise configurations.

- The number of errors the SCG produced using the random configurations.

- The time taken for the pairwise generator to create all of its configurations.

- The time taken for the pairwise generator to create all of its configurations and for the SCG to process these configurations.
- The time taken for the random generator to create a given number of configurations.
- The time taken for the random generator to create a given number of configurations and for the SCG to process these configurations.
- The number of parameter definition elements covered by each generator.

These metrics were recorded over a number of nightly builds. The data was collected and analysed and it was determined that both generators were successful at finding issues. We also looked at the overhead introduced by the pairwise generator when creating the configurations and the impact of this overhead when considering the time it takes to run the configurations through the SCG. We compared these times with the times needed by the random generator. We saw that there was an overhead introduced by the pairwise generator when creating configurations, but when considering the SCG time, this overhead decreased dramatically.

1.5 Outline

Section 2 of this document provides more detailed background information for the thesis. Section 3 provides details on how the research was conducted. Section 4 describes the architecture containing the random and pairwise generators developed as part of the thesis. Section 5 describes the algorithms developed for the random and pairwise generators. Section 6 describes the experiments conducted and their results. Section 7 presents academic literature related to this thesis. Section 8 provides a conclusion of the thesis and presents ideas on future work to be conducted following this thesis and limitations to this thesis. The references used throughout the report are provided in Section 9.

2. Background

This chapter provides background information about AUTOSAR, its layered software architecture and the Basic Software modules. It also contains detailed information about the parameter definition files and the limitation files which we use as an input to our generators. Information regarding the output of our generators, the configuration files, is also provided in this chapter. We discuss the difference between a valid and a working configuration and the expected results of the Source Code Generators. We provide background information about software product lines and one of the testing techniques that we decided to apply in this thesis, called pairwise.

2.1. AUTOSAR Layered Software Architecture

As mentioned briefly in the introduction and described in figure 1, the software architecture of AUTOSAR has on the highest level of abstraction an Application Layer, a Runtime Environment Layer and a Basic Software Layer. They run on a microcontroller (usually 16 or 32 bit), where internal devices such as Internal EEPROM, Internal CAN controller and Internal ADC (Analogue Digital Converter) are located (AUTOSAR Layered Software Architecture, 2011).

In the Application Layer the architecture changes from 'layered' to 'component style'. An application running on the AUTOSAR infrastructure consists of interconnected AUTOSAR Software Components (SW-C), therefore the AUTOSAR Software Components are located in the Application Layer. These AUTOSAR Software Components have standardized AUTOSAR interfaces, they can be mapped to an Electronic Control Unit (ECU) and each one encapsulates a part of the application's functionality. An example of an AUTOSAR Software Component is the Automatic Light Control which together with other interconnected AUTOSAR Software Components can be used to control the vehicle lights depending on the luminosity coming from outside the vehicle. A special type of AUTOSAR Software Components is the Sensor/Actuator Software Component, which is dependent on different sensors or actuators (AUTOSAR Technical Overview, 2008).

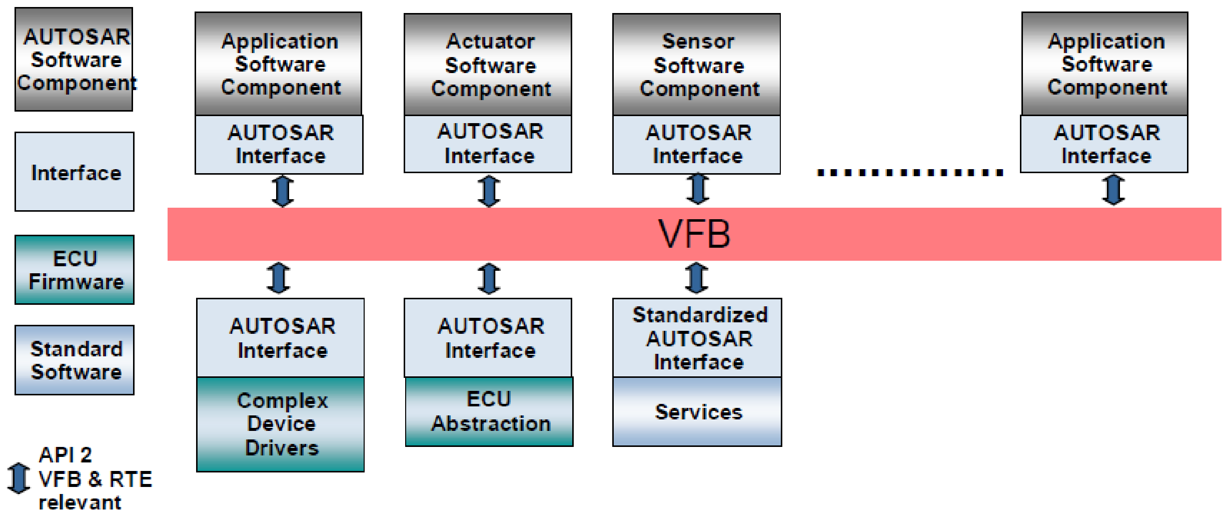


Figure 3 - AUTOSAR SW-C and AUTOSAR Services connected to the VFB (AUTOSAR Technical Overview, 2008).

The implementation of the AUTOSAR Software Components is independent of the underlying hardware. This independence is realized by the Virtual Functional Bus (VFB) which represents the abstraction of the communication mechanisms between the AUTOSAR Software Components and also includes interfaces to the Basic Software Layer. It enables the virtual integration of the AUTOSAR Software Components in an early phase of the development process. Figure 3 shows how different AUTOSAR Software Components and different parts of the Basic Software such as Complex Device Drivers, the ECU Abstraction and AUTOSAR Services are connected to the Virtual Functional Bus. While the ECU Abstraction and the Complex Device Drivers are ECU specific, the AUTOSAR Services Interface is standardized (AUTOSAR Technical Overview, 2008).

The Runtime Environment implements the functionality of the Virtual Functional Bus on a specific ECU. It makes the communication between the AUTOSAR Software Components independent of the communication channels and mechanisms. The implementation of an AUTOSAR Software Component cannot use the communication layer directly and it cannot access the Basic Software directly. To communicate with other AUTOSAR Software Components, it uses ports and client-server or sender-receiver communication. To access the Basic Software it uses ports and AUTOSAR Interfaces. The responsibility of the RTE is to generate the appropriate APIs for allowing access for the AUTOSAR Software Components (AUTOSAR Technical Overview, 2008).

The Basic Software Layer is used to run the functional part of the software. As described in figure 4 the Basic Software Layer is divided into four different layers: Services, ECU Abstraction, Complex Drivers and Microcontroller Abstraction. The Services and the Microcontroller Abstraction layers contain standardized components, while the ECU Abstraction and Complex Drivers layers contain ECU specific components *(AUTOSAR Layered Software Architecture, 2011)*.

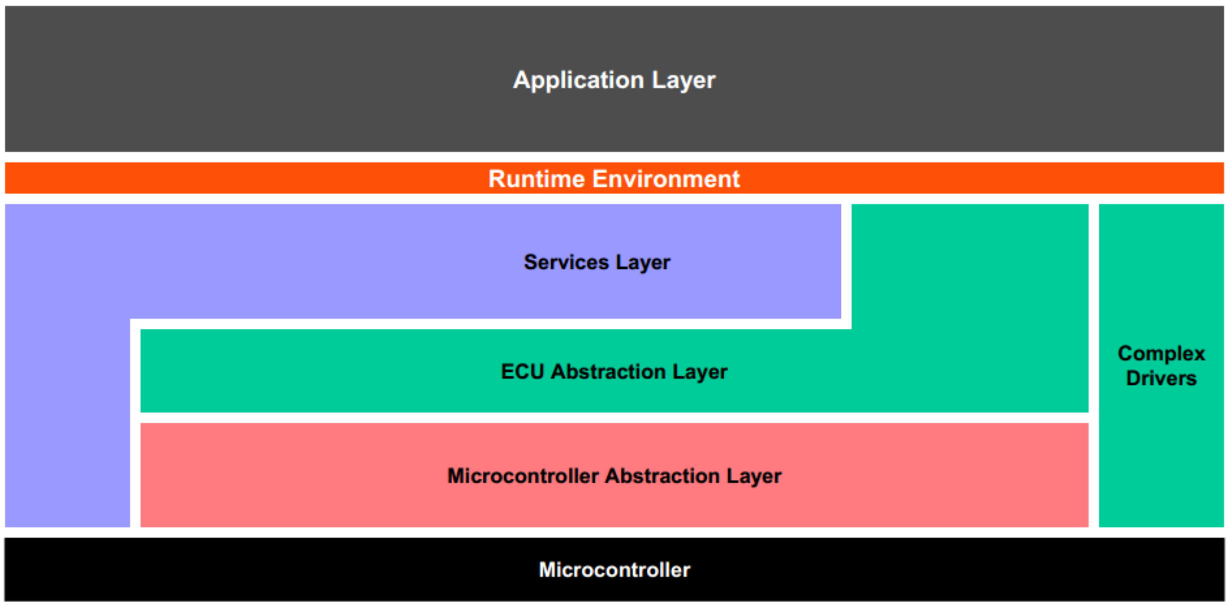


Figure 4 - The Layers of the Basic Software (AUTOSAR Layered Software Architecture, 2011).

The Services Layer is the most important layer for the application software as it provides basic services for the application and Basic Software modules to abstract the ECU hardware and the microcontroller from the layers above them. It offers operating system functionality, memory management services, vehicle network communication and management services, diagnostic services and mode management (AUTOSAR Technical Overview, 2008). As described in figure 5, the Services Layers is further divided into three functional groups: System Services, Memory Services and Communication Services.

The ECU Abstraction Layer is used to abstract the ECU Layout from the above layer. It abstracts the access to peripherals and devices and offers an API to access them independently of their location and their connection to the microcontroller (AUTOSAR Technical Overview, 2008). The ECU Abstraction Layer is further divided into four functional groups: Onboard Device Abstraction, Memory Hardware Abstraction, Communication Hardware Abstraction and I/O Hardware Abstraction.

The Complex Drivers Layer offers the possibility to integrate special functionality such as drivers for AUTOSAR external devices and it meets the special requirements for handling complex sensors and actuators. It can also be used to implement drivers for hardware that is not supported by AUTOSAR (AUTOSAR Technical Overview, 2008).

The lowest layer of the Basic Software is the Microcontroller Abstraction Layer (MCAL). It makes the upper layers independent on the microcontroller. It contains drivers that have access to the internal peripherals of the microcontroller and to the external devices mapped to the microcontroller (AUTOSAR Technical Overview, 2008). As described in figure 5 the MCAL is further divided into four functional groups: Microcontroller Drivers, Memory Drivers, Communication Drivers and I/O Drivers.

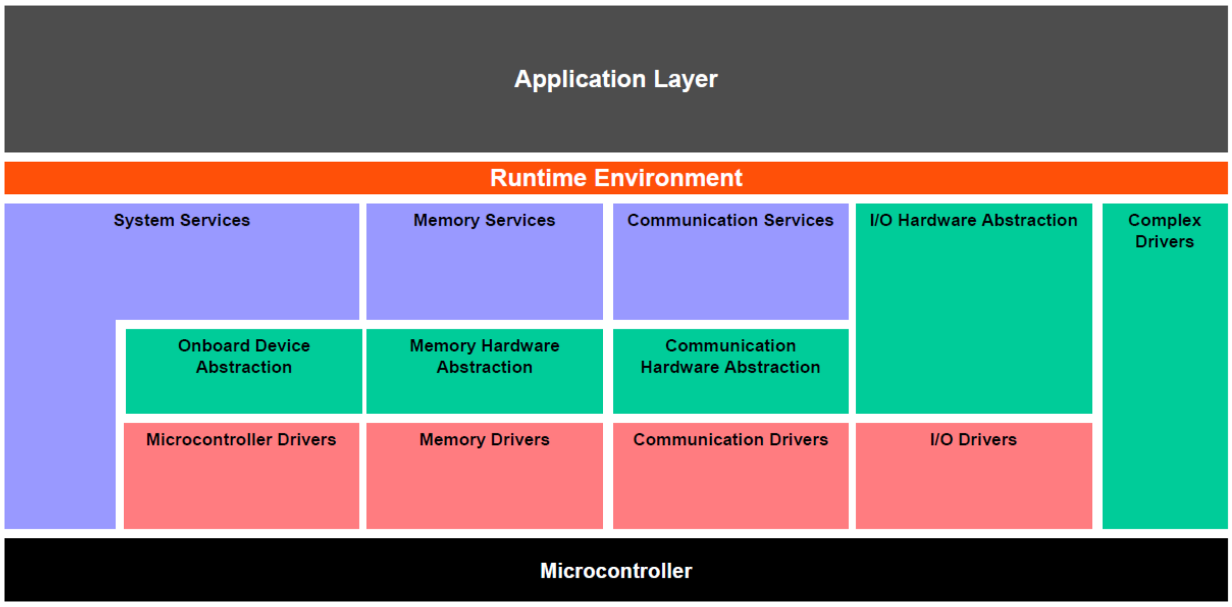


Figure 5 - The functional groups of the Basic Software (AUTOSAR Layered Software Architecture, 2011).

The functional groups described in figure 5 are further divided into Basic Software modules. There are over 70 modules defined, as described in figure 2. In figure 6 some of these modules are represented and grouped into functional groups.

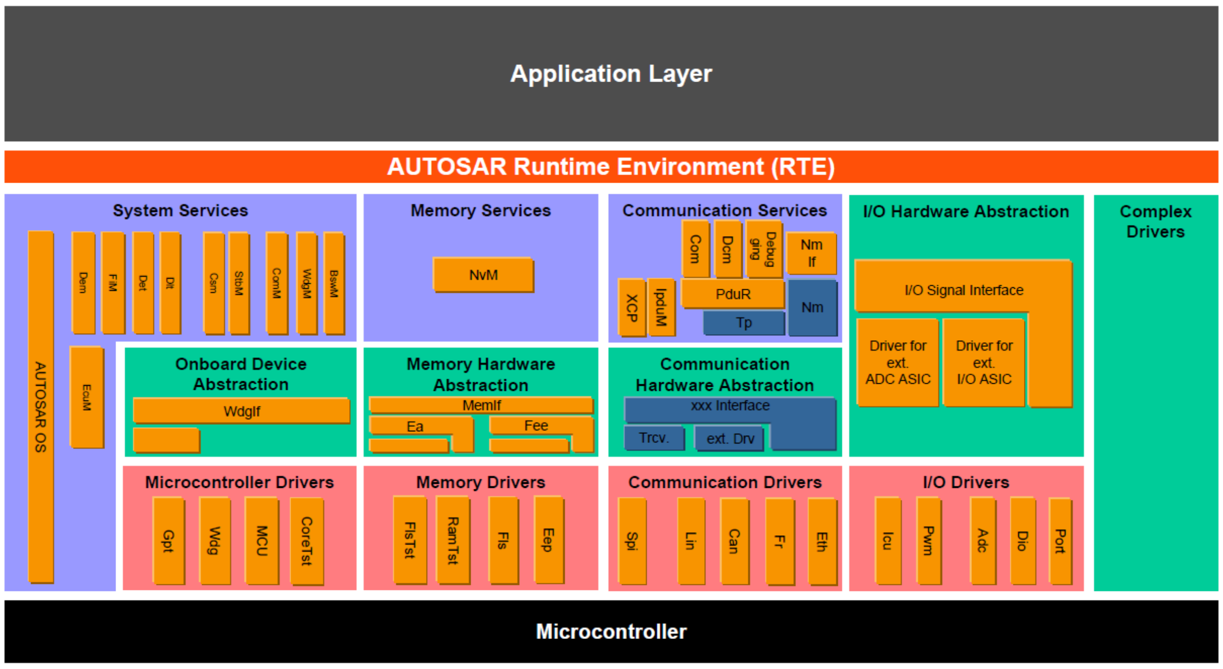


Figure 6 - Basic Software modules grouped into functional groups (AUTOSAR Layered Software Architecture, 2011).

Each one of the Basic Software modules is defined by a specific parameter definition file. The parameter definition file is an ARXML file and it conforms to an AUTOSAR standardized metamodel, called ECU Configuration Parameter Definition metamodel. The Basic Software modules are configured by creating a configuration file. This configuration file is also an ARXML file and it conforms to another AUTOSAR standardized metamodel, called ECU Configuration Parameter Values metamodel (AUTOSAR ECU, 2011).

Before continuing with the structure of these two metamodels, we present an example of a Basic Software module, called Development Error Tracer (Det). This module is used during the development of Software Components and other Basic Software modules to detect and trace errors. It evaluates the messages received from the other modules and components (AUTOSAR Det, 2011).

The structure of the parameter definition for a vendor specific Det module is described in figure 7 and it consists of containers and parameters. A container from the parameter definition is used to group several other elements such as sub-containers, parameters or references.

The first container in this example is called General and it contains the generic parameters of the Det module. The parameter Version API is a boolean parameter which is used to enable or disable the API to read the information about the modules version (AUTOSAR Det, 2011). This parameter is mandatory and it can only have the values 0 (false, disable) and 1 (true, enable). Another parameter contained by General is called Dlt. This parameter is optional and is also a boolean parameter with the possible values 0 and 1. If it is present and enabled (set to 1) the Det requires the interface of the Dlt module and forwards its call to a function called Dlt_DetForwardErrorTrace (AUTOSAR Det, 2011). The other two parameters included in the General container, Platform and ForeignModule, are vendor specific parameters so they are not mentioned in the AUTOSAR specification of the Det module (AUTOSAR Det, 2011).

The second container in this example is called Notification and it contains the configuration of the notification functions (AUTOSAR Det, 2011). The only parameter contained by Notification is called ErrorHook and it is a function name parameter. The multiplicity of this parameter is infinite, so a configuration may contain as many ErrorHooks as needed. They represent a list of functions to be called by the Det module when it is required to report an error (AUTOSAR Det, 2011).

Notes:

- The names of the containers and parameters of the Det module have been slightly modified for this example in order for the reader to get a better understanding of this module. For example the Dlt parameter is called DetForwardToDlt in the Det specification.

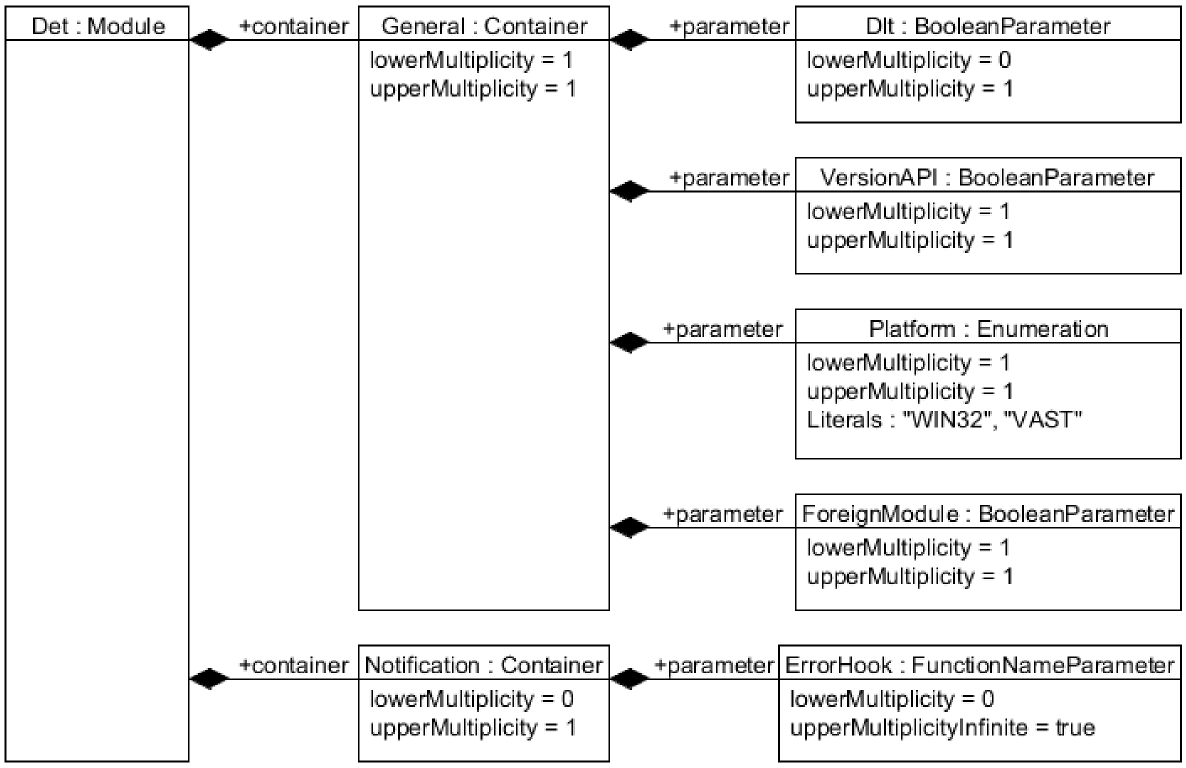


Figure 7 - Development Error Tracer Parameter Definition Structure.

To configure this Det module, a configuration file needs to be created. To be valid, this configuration has to conform to the parameter definition previously described. It has to contain one and only one General container which has to have exactly one VersionAPI, one Platform and one ForeignModule parameter. The General Container may also contain maximum one Dlt parameter. The values of these parameters have to be assigned according to their value range. The configuration file may also include a Notification container which may have as many ErrorHooks as needed. A valid configuration is described in figure 8.

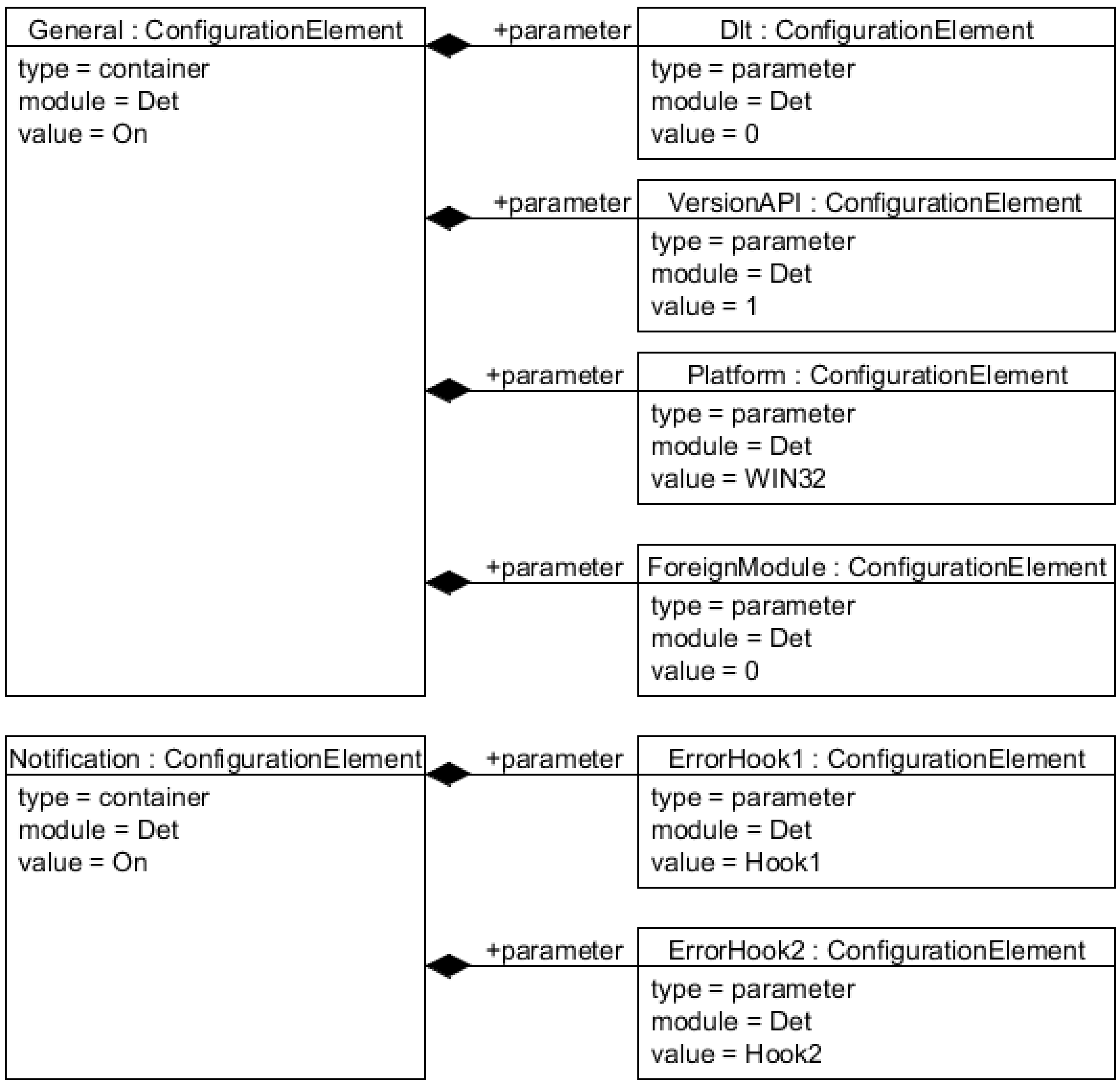


Figure 8 - Development Error Tracer Configuration Structure.

2.2. ECU Configuration Parameter Definition Metamodel

As described in figure 9, the top-level structure of a parameter definition file (ECU Configuration Parameter Definition) contains a Definition Collection (EcucDefinitionCollection) or a Module (EcucModuleDef) which are both derived from an AUTOSAR Element (ARElement). A Definition Collection contains at least one Module to be defined in the parameter definition file. Each Module has at least one Container (EcucContainerDef) whose structure is defined in figure 10.

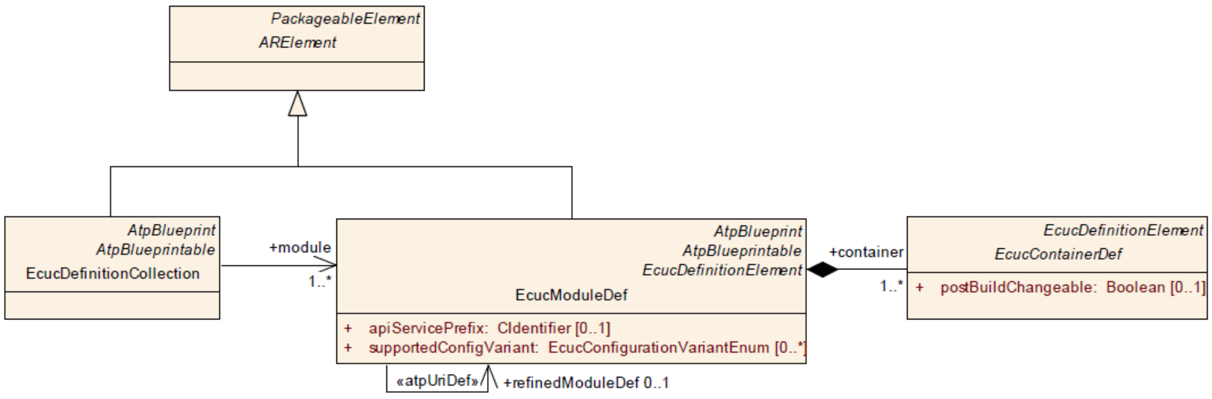


Figure 9 - The top-level structure of a ECU Configuration Parameter Definition (AUTOSAR ECU, 2011).

2.2.1. Containers

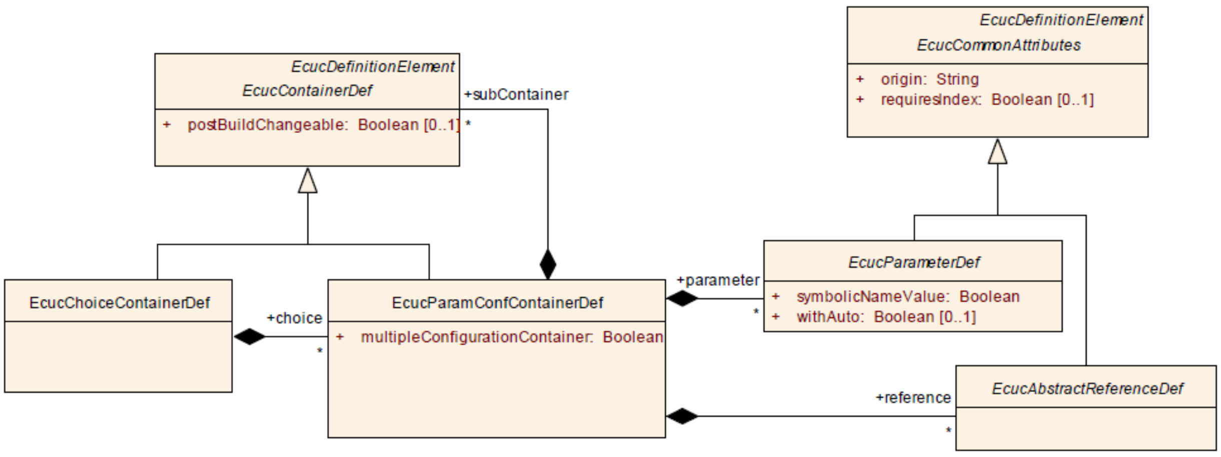


Figure 10 - The structure of a container's definition (AUTOSAR ECU, 2011).

As described in figure 10, an EcucChoiceContainerDef and an EcucParamConfContainerDef are both derived from a Container (EcucContainerDef). An EcucChoiceContainerDef allows for the selection of one and only one EcucParamConfContainerDef from different choices. An EcucParamConfContainerDef is a logical grouping of many Sub-Containers (EcucContainerDef), many Parameters (EcucParameterDef) and many References (EcucAbstractReferenceDef).

A Parameter is an attribute (EcucCommonAttributes) of a certain type (e.g. boolean, integer, float, string, function name, etc.), while a Reference is an attribute (EcucCommonAttributes) that is used to create a link to another EcucParamConfContainerDef.

In order to specify how many times a specific container, parameter or reference may be included in a configuration, both the EcucContainerDef and EcucCommonAttributes inherit from the EcucDefinitionElement class. This class has attributes that can define a lower, a finite upper multiplicity and an infinite upper multiplicity. For example, a container can have a lower multiplicity set to 0 and an upper multiplicity set to 5, which means that the container is optional and that the maximum number of times this container can be included in a configuration is 5.

2.2.2. Parameters

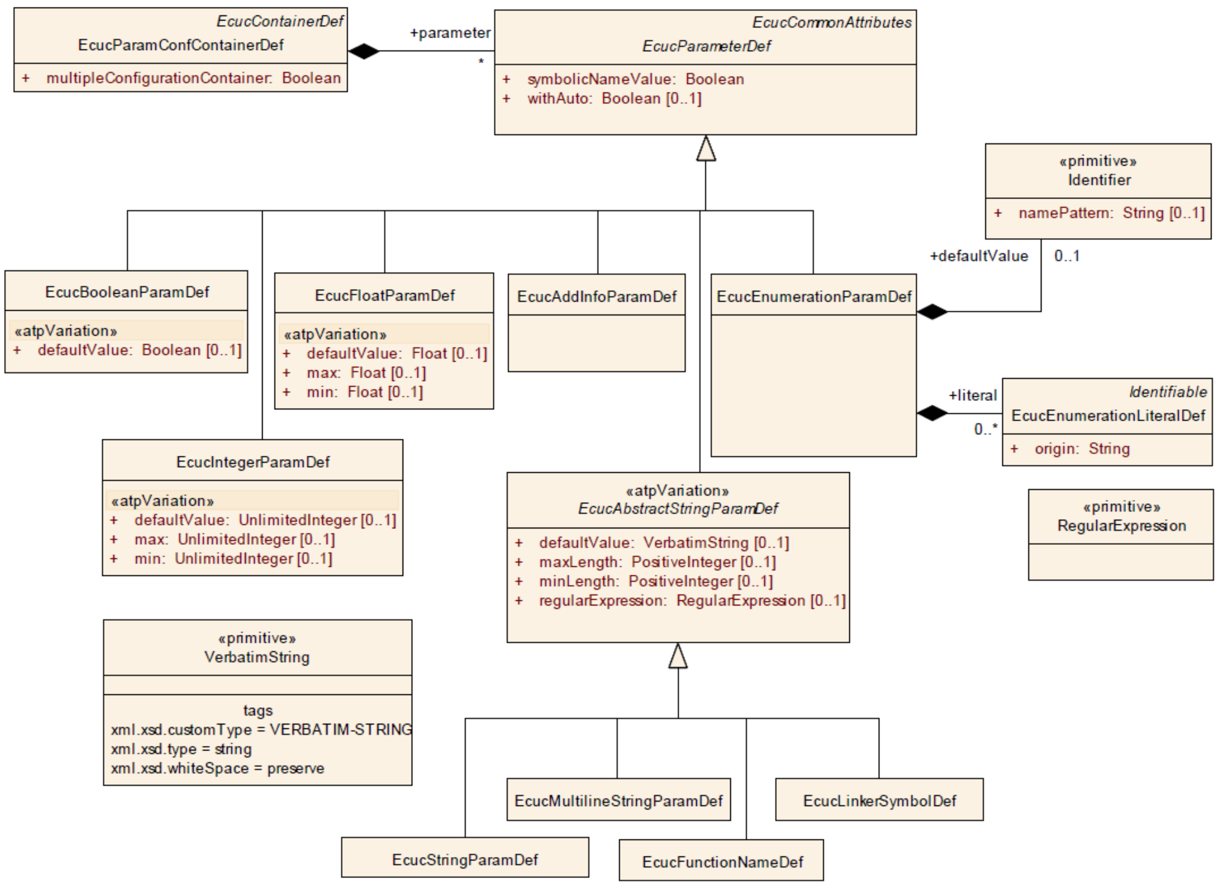

EcucContainerDef
EcucParamConfContainerDef
+ multipleConfigurationContainer: Boolean
+parameter
*
EcucCommonAttributes
EcucParameterDef
+ symbolicNameValue: Boolean
+ withAuto: Boolean [0..1]
«primitive»
Identifier
+ namePattern: String [0..1]
+defaultValue
0..1
EcucBooleanParamDef
«atpVariation»
+ defaultValue: Boolean [0..1]
EcucFloatParamDef
«atpVariation»
+ defaultValue: Float [0..1]
+ max: Float [0..1]
+ min: Float [0..1]
EcucAddInfoParamDef
EcucEnumerationParamDef
+literal
0..*
Identifiable
EcucEnumerationLiteralDef
+ origin: String
EcucIntegerParamDef
«atpVariation»
+ defaultValue: UnlimitedInteger [0..1]
+ max: UnlimitedInteger [0..1]
+ min: UnlimitedInteger [0..1]
«atpVariation»
EcucAbstractStringParamDef
+ defaultValue: VerbatimString [0..1]
+ maxLength: PositiveInteger [0..1]
+ minLength: PositiveInteger [0..1]
+ regularExpression: RegularExpression [0..1]
«primitive»
RegularExpression
«primitive»
VerbatimString
tags
xml.xsd.customType = VERBATIM-STRING
xml.xsd.type = string
xml.xsd.whiteSpace = preserve
EcucMultilineStringParamDef
EcucLinkerSymbolDef
EcucStringParamDef
EcucFunctionNameDef

Figure 11 - The structure of a parameter's definition (AUTOSAR ECU, 2011).

As described in figure 11, a parameter is contained in an EcucParamConfContainerDef and it can be a boolean (EcucBooleanParamDef), an integer (EcucIntegerParamDef), a float (EcucFloatParamDef), an addInfo (EcucAddInfoParamDef), a string (EcucAbstractStringParamDef) or an enumeration (EcucEnumerationParamDef). Each of these different types of parameters has attributes. For example, an integer and a float can have a minimum, a maximum and a default value. A string parameter can have a regular expression that it needs to conform to, a minimum length, a maximum length and a default value. An enumeration parameter has a list of literals it can choose its value from.

2.2.3. References

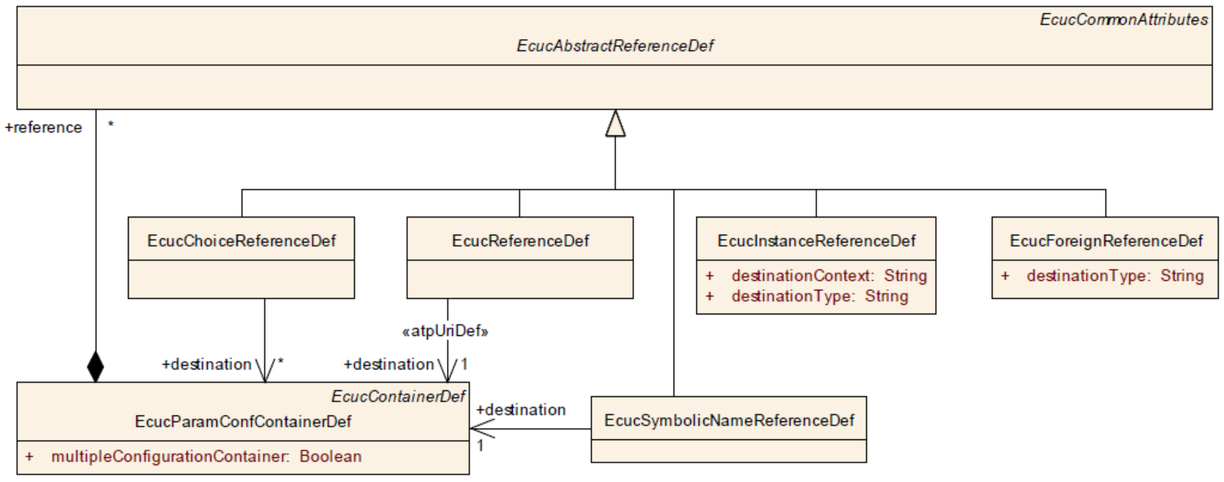


Figure 12 - The structure of a reference's definition (AUTOSAR ECU, 2011).

As described in 12, there can be five types of references and they are contained in an EcucParamConfContainerDef. EcucReferenceDef is a reference that has one destination to another ParamConfContainer within the same configuration. A choice reference (EcucChoiceReferenceDef) has multiple destinations to choose from and they are all referring to other ParamConfContainers within the same configuration. A symbolic name reference (EcucSymbolicNameReferenceDef) has a destination to a ParamConfContainer within the same configuration that contains a parameter with the symbolic name boolean value set to true. Only one of the parameters of a certain ParamConfContainer can have the symbolic name boolean value set to true. Elements outside the configuration can be referenced using instance references (EcucInstanceReferenceDef) or foreign references (EcucForeignReferenceDef). The foreign reference can be used when the referenced element has been specified in a different AUTOSAR template, while the instance reference can be used using the instanceRef semantics defined in the Generic Structure Template (AUTOSAR ECU, 2011).

2.3. ECU Configuration Parameter Values Metamodel

As described in figure 13, the top-level element of a configuration file (ECU Configuration Parameter Values) is the value collection (EcucValueCollection). It needs to reference the System description, provided as an ecuExtract. A value collection also references at least one configuration module (EcucModuleConfigurationValues), provided as an ecucValue. The configuration module is defined by one module from the parameter definition file. It contains at least one container (EcucContainerValue) which is defined by a container (EcucContainerDef) from the parameter definition file. The configuration module can also have a module description referring to a BSW Implementation.

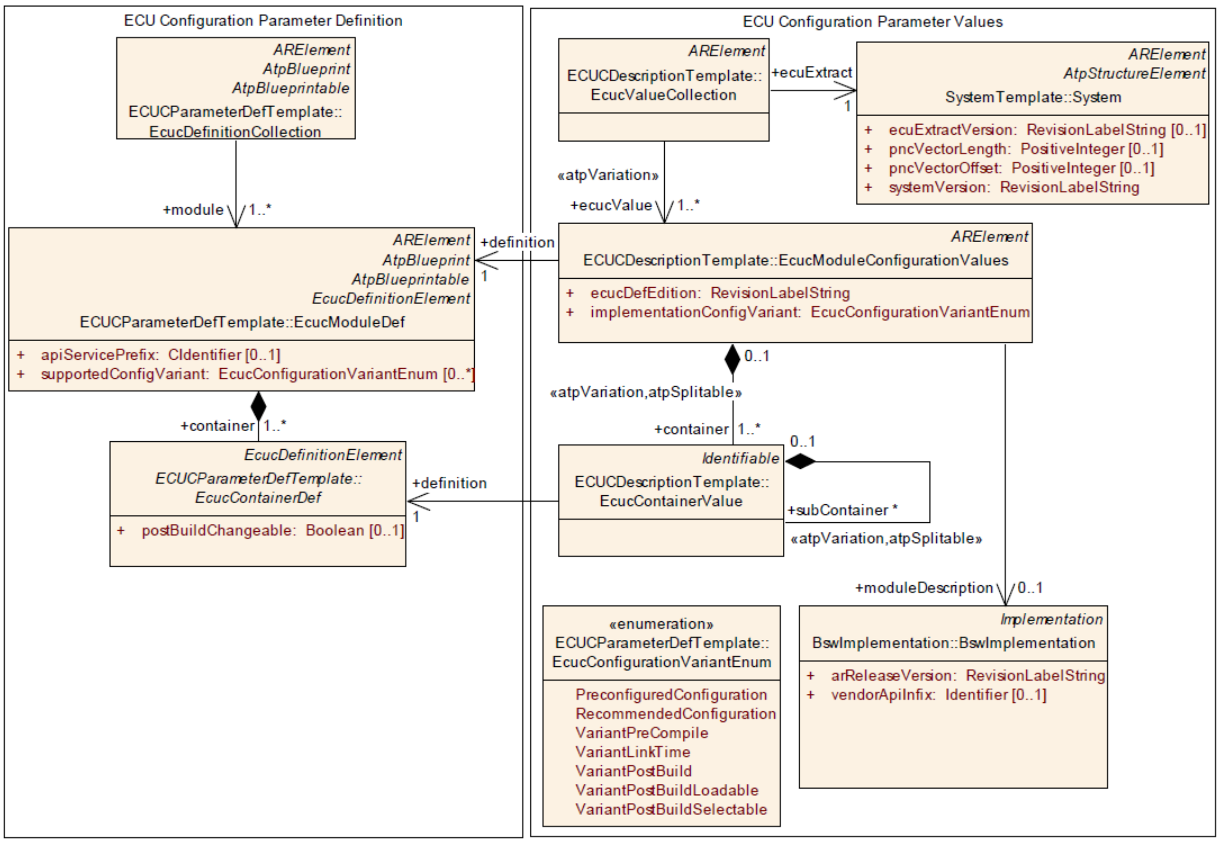

ECU Configuration Parameter Definition
ARElement
AtpBlueprint
AtpBlueprintable
ECUCParameterDefTemplate::
EcucDefinitionCollection
+module
1..*
ARElement
AtpBlueprint
AtpBlueprintable
EcucDefinitionElement
ECUCParameterDefTemplate::EcucModuleDef
+ apiServicePrefix: CIdentifier [0..1]
+ supportedConfigVariant: EcucConfigurationVariantEnum [0..*]
+container
1..*
EcucDefinitionElement
ECUCParameterDefTemplate::
EcucContainerDef
+ postBuildChangeable: Boolean [0..1]
+definition
1
+definition
1
ECU Configuration Parameter Values
ARElement
ECUCDescriptionTemplate::
EcucValueCollection
+ecuExtract
1
ARElement
AtpStructureElement
SystemTemplate::System
+ ecuExtractVersion: RevisionLabelString [0..1]
+ pncVectorLength: PositiveInteger [0..1]
+ pncVectorOffset: PositiveInteger [0..1]
+ systemVersion: RevisionLabelString
«atpVariation»
+ecucValue
1..*
ARElement
ECUCDescriptionTemplate::EcucModuleConfigurationValues
+ ecucDefEdition: RevisionLabelString
+ implementationConfigVariant: EcucConfigurationVariantEnum
0..1
«atpVariation,atpSplitable»
+container
1..*
0..1
Identifiable
ECUCDescriptionTemplate::
EcucContainerValue
+subContainer *
«atpVariation,atpSplitable»
+moduleDescription
0..1
«enumeration»
ECUCParameterDefTemplate::
EcucConfigurationVariantEnum
PreconfiguredConfiguration
RecommendedConfiguration
VariantPreCompile
VariantLinkTime
VariantPostBuild
VariantPostBuildLoadable
VariantPostBuildSelectable
Implementation
BswImplementation::BswImplementation
+ arReleaseVersion: RevisionLabelString
+ vendorApiInfix: Identifier [0..1]

Figure 13 - The top-level structure of an ECU Configuration Value (AUTOSAR ECU, 2011).

A container from a configuration can include the same type of elements as in the parameter definition file, i.e. sub-containers, parameters and references. The structure of a parameter (EcucParameterValue) in a configuration is described in figure 14. A parameter is contained in an EcucContainerValue and it can be a numerical (EcucNumericalParamValue), textual (EcucTextualParamValue) or addInfo parameter (EcucAddInfoParamValue). A numerical parameter is used for booleans, integers and floats, while a textual parameter is used for enumerations and strings. Each configuration parameter has its definition referring to a parameter (EcucParameterDef) from the parameter definition file.

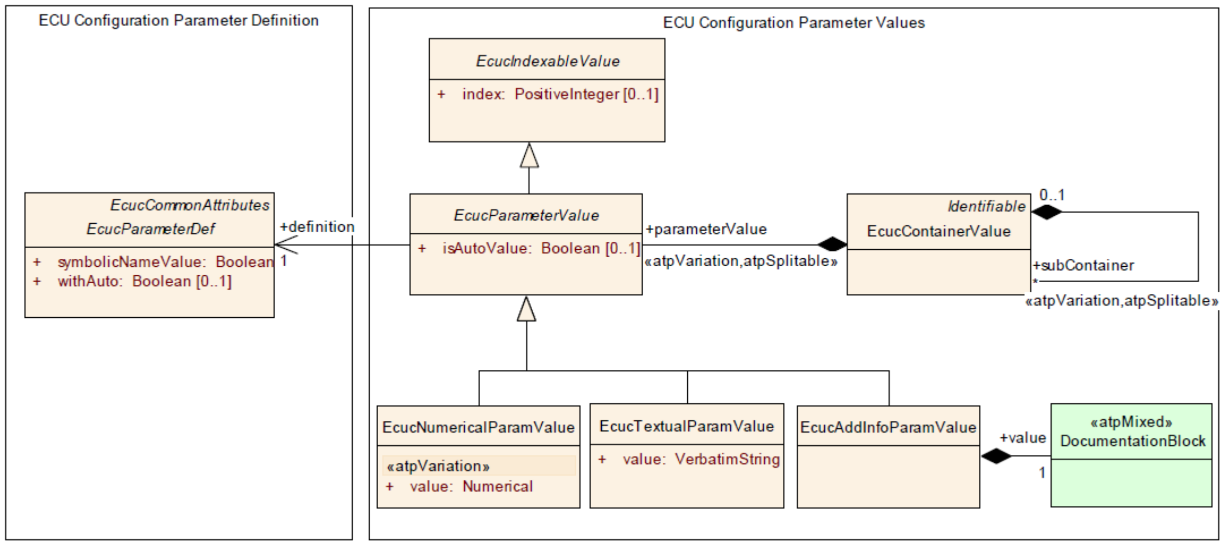


Figure 14 - The structure of a parameter in a configuration (AUTOSAR ECU, 2011).

According to figure 15, the references (EcucAbstractReferenceValue) of a configuration are also contained in an EcucContainerValue and they are of two types: EcucInstanceReferenceValue and EcucReferenceValue. The EcucInstanceReferenceValue is defined by an EcucInstanceReferenceDef from the parameter definition file, while the EcucReferenceValue can be defined by any of the other types of references from the parameter definition file, i.e. EcucReferenceDef, EcucChoiceReferenceDef, EcucForeignReferenceDef and EcucSymbolicNameReferenceDef.

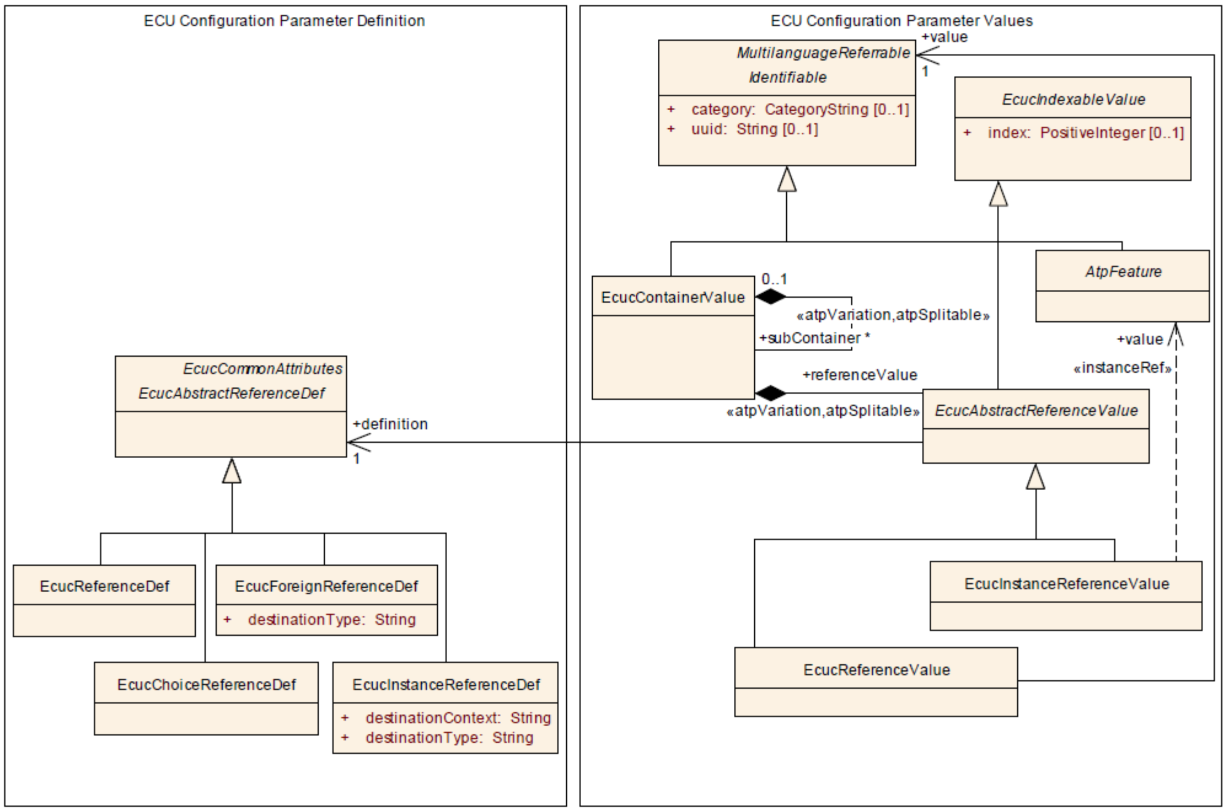


Figure 15 - The structure of a reference in a configuration (AUTOSAR ECU, 2011).

2.4. Limitation Files

A requirement from Mecel AB was to be able to limit the parameter definition file. Say for example we have a parameter called ErrorHook that has a lower multiplicity of 0 and an upper multiplicity of 65000. The tester responsible for the generators may not wish to run configurations with 65000 instances of ErrorHook. They may know that in reality 5 ErrorHooks are sufficient for testing. They can use the limitation definition to define the new multiplicity. Another common attribute to define limitations for is the value range that a parameter may contain. Limitations can be defined for all parameter and reference attributes. With the exception of the short name any part of these elements can be modified using the limitation file. For containers only the multiplicities need to be limited. The user may also make sure that an element is not configured by setting both its lower and upper multiplicity to 0.

The limitation definition conforms to the AUTOSAR XML schema and is a subset of the parameter definition file, i.e. the limitation file includes everything that is in the parameter definition file with the exception that it can exclude unmodified elements and the elements it does include can be modified.

2.5. Valid and Working Configurations

When discussing about the configuration files of a Basic Software module we make a difference between a valid and a working configuration. A valid configuration conforms to the ECU Configuration Parameter Values Metamodel and to the AUTOSAR XML schema. A working configuration, as defined by Mecel AB, is a valid configuration that satisfies all the AUTOSAR and vendor specific constraints defined for the Basic Software module. More details about these constraints can be found in the next section. A working configuration is a configuration that can be used by a Source Code Generator to generate code. A valid but not working configuration cannot be used to generate code, but it can be used to test if the Source Code Generator rejects it and prints a descriptive error message. More details about the Source Code Generators can be found in section 2.7.

2.6. Constraints

There are many different constraints for every module of the Basic Software Layer. These constraints can be vendor specific or they can be defined by AUTOSAR. Only some of the AUTOSAR/vendor defined constraints can be handled in the parameter definition files. For example, if a constraint is that the integer parameter called ‘Id’ needs to be configured for the container called ‘Event’, then the lower multiplicity of this parameter is set to ‘1’ in the parameter definition file. If this is the only constraint in a certain module, then a valid configuration, i.e. which conforms to the parameter definition file, is also a working configuration. Some vendor specific constraints can also be specified in a vendor specific parameter definition file or in the limitation file. For example, an enumeration parameter that can have as a value one of the literals ‘WIN32’ or ‘VAST’ could be constrained by a vendor that does not support one of the literals. In this case, the vendor could create a specific parameter definition file and remove the unsupported literal or it could use the limitation file to do the same thing.

The most interesting constraints are the ones that cannot be specified in a parameter definition or a limitation file, so they are specified in natural language in the AUTOSAR or vendor specific module specifications. For example, all instances of an integer parameter called ‘id’ could be constrained to have consecutive values, i.e. the configuration parameter ‘id_first’ has the value 0, ‘id_second’ has the value 1 and so on.

2.7. Source Code Generators

There is one vendor specific SCG (Source Code Generator) for each of the modules of the Basic Software Layer. As described in figure 16, a SCG takes in a configuration of a specific module and its parameter definition file. The SCG checks if all the constraints have been satisfied for the given configuration and only if this test passes it generates code in format of .c and .h files. This generated source code, together with a static source code, is used to build executables. An invalid configuration is rejected straight away by the SCG, while a valid configuration needs to pass the constraints check in order to be considered a working configuration. From here there are only two possible scenarios. The most probable one is that the SCG generates source code, while the other scenario is that the SCG encounters problems and throws an error or it may just crash. Considering the first scenario, the generated source code is then tested and compiled to make sure that it is valid. For this study, of the unit tests created by Mecel AB there was no unit test suite for verifying the source code we produce so this verification was considered as out of scope.

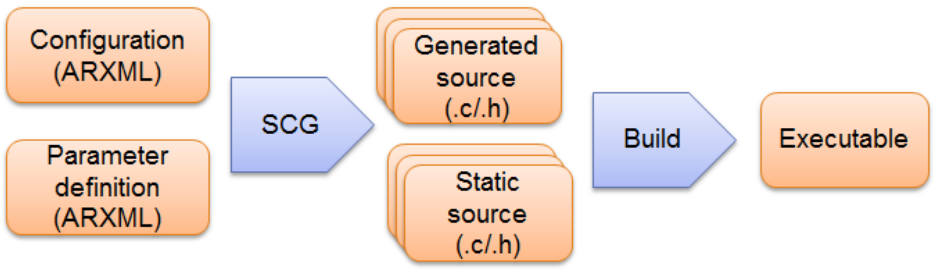


Figure 16 - Source Code Generator Input and Output.

2.8. Software Product Lines

As stated in the Introduction section, the use of software product lines involves identifying common features that can be used across an organization's product range. The use of a common base allows for increased quality and faster development time. When a product requires a feature that is specific to just that product a variability point is introduced. Each variation point increases combinatorial complexity and the adding of variation points is carefully controlled (McGregor, 2010). A simple, yet incomplete, example of a product line for a car is described in figure 17, where a car has a common feature representing an engine and a variable feature representing a navigation system. The engine has four common features for ignition, induction, emission and compression. The navigation system has only one variable feature called Voice command. Any car product resulting from this product line has to have at least the engine with all it common features and it may include a navigation system with or without a voice command.

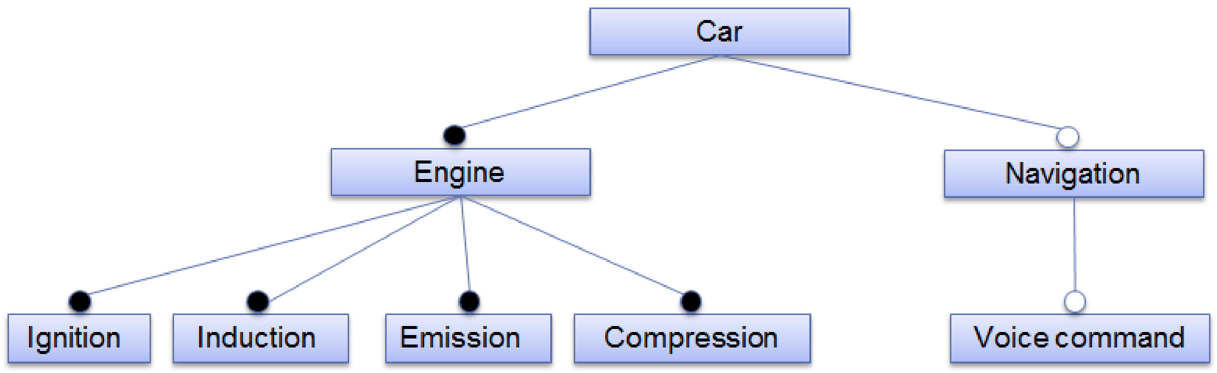


Figure 17 - Software product line example.

The parameter definition files also contain mandatory and optional elements. For example, in figure 18 we have a module called ‘Det’ that has one mandatory container called ‘General’ and one optional container called ‘Notification’. The mandatory container includes 4 mandatory parameters, while the optional container includes an optional parameter. Comparing this example with the one described in figure 17, we can conclude that a parameter definition file can be considered as a self-contained software product line.

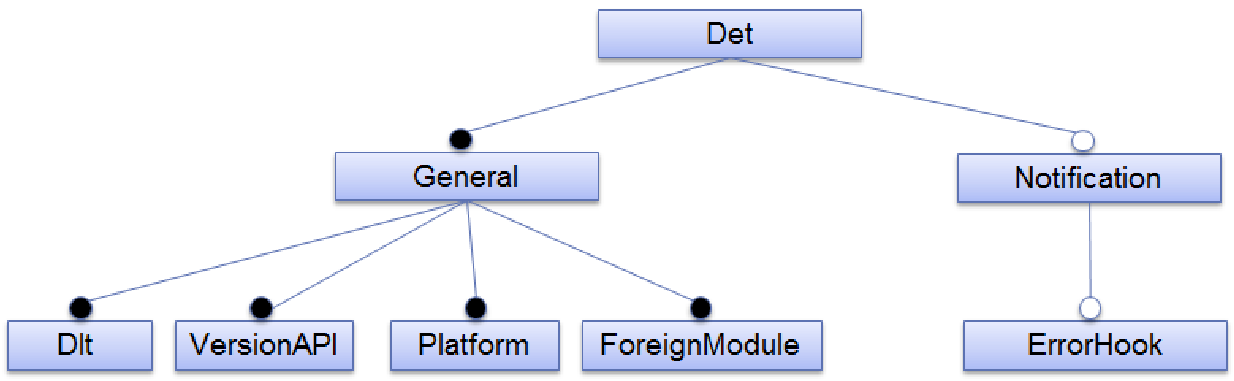


Figure 18 – The Det module as a software product line.

One technique used for testing software product lines is pairwise testing which we also apply in this thesis.

2.9. Pairwise Testing

Considering a system that contains multiple parameters which can have many different values, a test case for this type of system includes all or only some of the parameters of the system, each one having a value assigned to it. A medium size system with hundreds of parameters can have a very large number of test cases. Generating and testing all these test cases require a large amount of allocated resources and time (Cohen et al., 1996).

The combinatorial design method is a way to reduce the number of test cases of a system. This method generates tests that cover all pairwise, triple up to n-way combinations of parameters of a system (Cohen et al., 1996). The n-way testing requires that for each set of n parameters, every combination containing the values of these n parameters is covered at least once in a test case (Lei et al., 2002).

Pairwise testing, i.e. 2-way testing, is one case of the n-way testing strategy, where n is equal to 2. Pairwise testing requires that for each pair of parameters of a system, every combination containing the values of these pairs is covered in at least one test case (Lei et al., 2002). The pairwise approach is discussed in many articles related to software testing (Cohen et al., 1996) (Colbourn et al., 2004) (Czerwonka, 2008) (Lei et al., 2002) (Williams, 2000). Several algorithms for implementing this approach have been published. There are two main pairwise techniques that can be applied when generating test cases. These are the “All Pairs” approach and the “Orthogonal and Covering Arrays” approach.

In this thesis, we decided to mainly adapt one of the pairwise techniques to our requirements. This uses the “All Pairs” approach and is called in-parameter-order, or IPO (Lei et al., 2002).

3. Research Methods

In this chapter we discuss about the case study as a research method and about the literature study.

3.1. Research Methods

For this research, we chose to conduct a case study. This section describes what a case study is, what other research methods exist and why we chose to use a case study as the research method. This section is mostly based on Robert K. Yin's book (Yin, 2009).

A case study is one of the several methods of conducting a research. Other research methods are survey, archival analysis, history, experiments and action research. The case study is described in Yin's book as a way to "illuminate a decision or set of decisions: why they were taken, how they were implemented, and with what result." A case study is, according to the same author, an "empirical inquiry that investigates a contemporary phenomenon in depth and within its real-life context".

For this study we aimed to investigate automatic generation of AUTOSAR configurations, and we planned to discover why it is needed, how we can implement it and what the results are of the implementation in a real-life context, at Mecel AB. There are variations between case studies as a research method. Yin's book describes three types of case studies used for research:

1. Explanatory or causal case studies.
2. Descriptive case studies.
3. Exploratory case studies.

An exploratory case study tendency is to find out what can be done, by seeking ideas or generating new ones and by testing new hypotheses (Runeson et al., 2009). Among the three types of case studies, the exploratory one is what is best suited to this thesis and it is the one we used as the research method for this study.

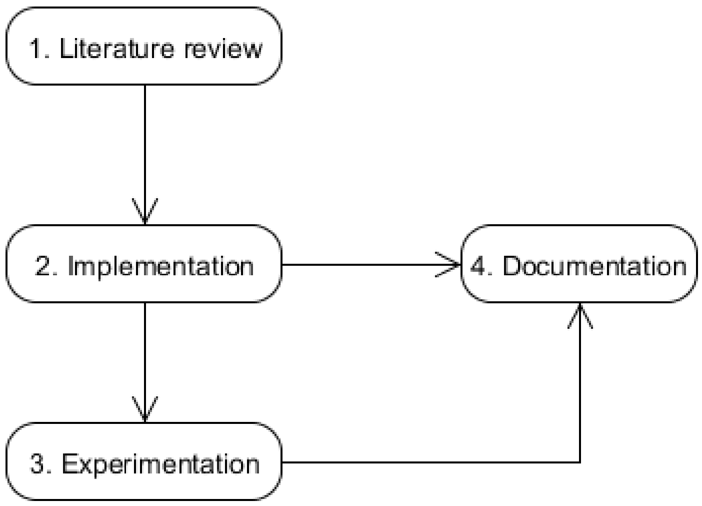


Figure 19 - Research steps.

Figure 19 illustrates the steps we went through during this study. The first step included in our process was Literature review. In order to gain the domain knowledge that was necessary to conduct this case study, we had to read related literature papers. The University of Chalmers library was our main resource for finding these papers. We first started to look for literature containing keywords such as: integration testing, random testing, test input generation, software product line testing, pairwise testing and pairwise algorithms. The papers we read gave us enough domain knowledge about efficient techniques to generate test cases. Next, we had to gain knowledge about AUTOSAR, the Basic Software modules, the parameter definition and configuration files. The main source we used to gather this data was the AUTOSAR website (AUTOSAR Home, 2012).

The next steps were to implement the solutions and conduct experiments to compare them. The last step described in this diagram was done continuously throughout our process and it involved writing this document.

4. Architecture

This chapter describes the architecture of the system. The architecture is represented as a diagram showing the existing artefacts and components used as part of this thesis and the artefacts and components developed over the course of the thesis and how these interact. The diagram is followed by a more detailed explanation of the architecture. At a high level we have two separate generators, the random generator and the pairwise generator. The random generator takes in an AUTOSAR parameter definition(s) and creates an internal representation of an AUTOSAR configuration that relates to one AUTOSAR BSW configuration. The random generator randomly chooses which elements to add by randomising the multiplicities and values of the elements. The pairwise generator also takes in an AUTOSAR parameter definition(s) but it creates a table where each row is an internal representation of an AUTOSAR configuration. The configuration creator takes the internal representation of the AUTOSAR configurations and produces an AUTOSAR BSW configuration for each. These configurations are then provided as input to the SCG for the purpose of testing.

4.1. Architecture Description

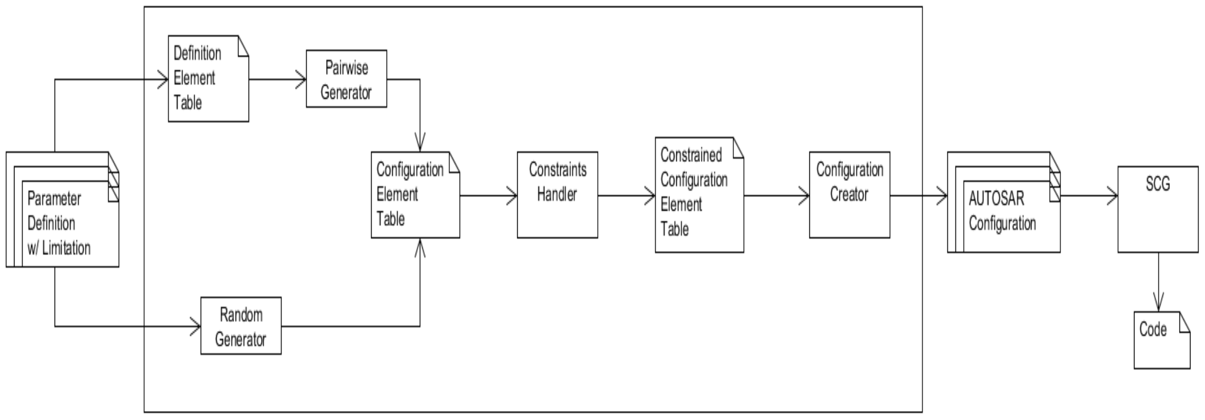


Figure 20 - Overall Architecture.

Figure 20 shows the complete flow of the system. The artefacts and components within the box were developed as part of the thesis work. The artefacts and components outside of the box provide the inputs for our generators and show how the results of the generators are used.

The **parameter definition** contains all the elements for the module(s) that are required by the generators. The parameter definition conforms to the AUTOSAR schema. An example of a parameter definition file can be seen in Appendix A. Multiple parameter definitions can be provided. There are two types of parameter definition, primary and secondary. Primary parameter definitions are those which the user wants to generate configurations for. Secondary parameter definitions are those which are referenced by the primary definitions and which do not require complete generation.

Primary parameter definitions can be limited for the purpose of generation. This is used to reduce the scale of what is being generated. For example if a container has an upper multiplicity as a value so high as to cause the state space to become infeasible to generate within a realistic time frame, then these values can be changed using a limitation definition file. The parameter definition and limitation definition are merged to create a merged parameter definition with the limitations applied. The definitions are merged by finding matching elements in the limitation definition to those that appear in the parameter definition. Only elements that require limitation are included in the limitation definition so if a matching element is found then the element in the limitation definition is given precedence over the corresponding parameter definition element and it is this element that will appear in the merged parameter definition. This merged parameter definition is provided to the generators. Multiple merged parameter definitions can be provided to the generators as the primary parameter definitions.

The **random generator** takes in the merged parameter definitions for all primary modules and the parameter definitions for the secondary modules. It applies the random algorithm and produces one row in a table of Configuration Elements that will be converted to an ARXML configuration.

A **Configuration Element** is an internal representation of an element that can later be added to the configuration ARXML file. The Configuration Element contains its value and its parameter definition information. The parameter definition information consists of the module it belongs to, its short name, its parent, its children, its type (e.g. parameter, reference, container) and its index. The index relates to where it should be positioned in the Configuration Element row. If the Configuration Element is a reference, then information is also stored about the referenced element and if the reference is in a secondary module.

If multiple primary definition files are provided as input each module will be run through the random algorithm. If one of the primary parameter definitions references a container in a secondary parameter definition then Configuration Elements need to be included for the secondary module. A complete representation of the secondary module is not required, just the referenced container, its parameters and its parent hierarchy. The algorithm used by the random generator is described in section 5.2.

The **pairwise generator** has the same requirements with respect to the primary and secondary parameter definitions. The data from the primary parameter definitions is converted into an internal structure, the **Definition Element Table**, in order to prepare the elements in a suitable manner for pairwise generation. Unlike the random generator which produces one row of Configuration Elements corresponding to one configuration, the pairwise generator produces multiple rows of Configuration Elements, each corresponding to one configuration. The pairwise algorithm is described in the section 5.3

Based on Mecel AB requirements we define two types of configurations. The first is a **valid configuration.** A valid configuration is one that conforms to the structure defined in the parameter definition and also to the AUTOSAR schema.

A **working configuration** is a configuration that is valid and conforms to additional constraints not available in the parameter definition. These constraints are defined by AUTOSAR and Mecel AB and when a developer is creating configurations manually they ensure that their configurations do not violate the constraints. The following is an example of a type of constraint. The example contains a mandatory container C1 with two boolean parameters P1 and P2. It also contains and an optional container called C2. An example of a constraint could be that in order for C2 to be enabled P2 must be set to true.

- Container1: C1 (mandatory container)
    - Parameter1: P1
    - Parameter2: P2
- Container2: C2 (optional container)

The Configuration Element Table that is produced by the random and pairwise generators contains rows that correspond to valid configurations. In order to turn these configurations into working configurations the Configuration Element Table is provided as input to the **constraints handler**. Each module has its own constraints handler which has the ability to identify violated constraints and update a row in the Configuration Element Table to ensure that the constraints are no longer violated. In this thesis we have implemented this for the FiM module and any of the rows the FiM constraints handler updates have no effect on the pairings covered by the pairwise algorithm.

The **configuration creator** takes in a table containing rows of Configuration Elements, the **Configuration Element Table**. Each row corresponds to one configuration. As a Configuration Element contains the value, type and module of the element and its parent/child, reference and choice information the configuration creator can loop through each element in a row and write to ARXML maintaining the correct relationships, types and values. The resulting AUTOSAR configuration conforms to the AUTOSAR schema.

When configurations have been created by the configuration creator they are then run through a **source code generator (SCG)**. The source code generator is the system under test. Each module has its own SCG. It should accept valid configurations without crashing. A crash is an unhandled exception that is thrown when validating the input files or when generating code from the configuration.

Internally the SCG has two main components. The first component runs the AUTOSAR and Mecel AB defined constraints against the configuration to make sure that it is a working configuration. This constraint checking component can be tested by providing working and valid configurations. When the SCG has determined the configuration is a working configuration it passes it to the second component which generates C code from the configuration. This second component can only be tested by working configurations.

The constraints checker in the SCG identifies three types of violations. The first are warnings, a configuration can be a working configuration and have warnings. Warnings do not prevent the SCG from generating code. The second type of violation is errors. Working configurations do not contain errors. The constraints component will continue to run even in the presence of errors so valid configurations can test this component fully. Errors are accumulated and if present stop the SCG from trying to generate code. The errors are then displayed.

The third and final type of violation is critical errors. These critical errors will stop the constraints component immediately. Working configurations are free from critical errors. Valid configurations that contain critical errors will only test the SCG constraints component up to the point that the critical error is encountered.

The SCG produces **code** when it completes successfully for a working configuration. Currently, of the unit tests provided by Mecel AB there is no unit test suite for verifying the code we produce so this verification is out of scope for the thesis.

5. Algorithms

This section describes the algorithms used in the implementation part of this project. Both algorithms are described using a common example.

5.1. Example

This section describes the parameter definitions of two modules. The modules we chose had to be complete enough to contain many of the different types of elements, but also quite compact and easy to understand. Some of the Basic Software modules are small enough for this example, but they do not contain a complete set of different elements. For example, the Det module (Development Error Tracer) does not contain elements such as Choice Containers or References. The larger modules, such as the Dcm module (Diagnostic Communication Manager), are far too complex to be used as an example in this chapter.

We decided to create our own modules called Dsm (Diagnostic Sample Module) and Drm (Diagnostic Referenced Module). The Dsm module is the primary module for this example, while Drm is used as a secondary module because it is referenced by Dsm. The structure of the two modules is described in figure 21.

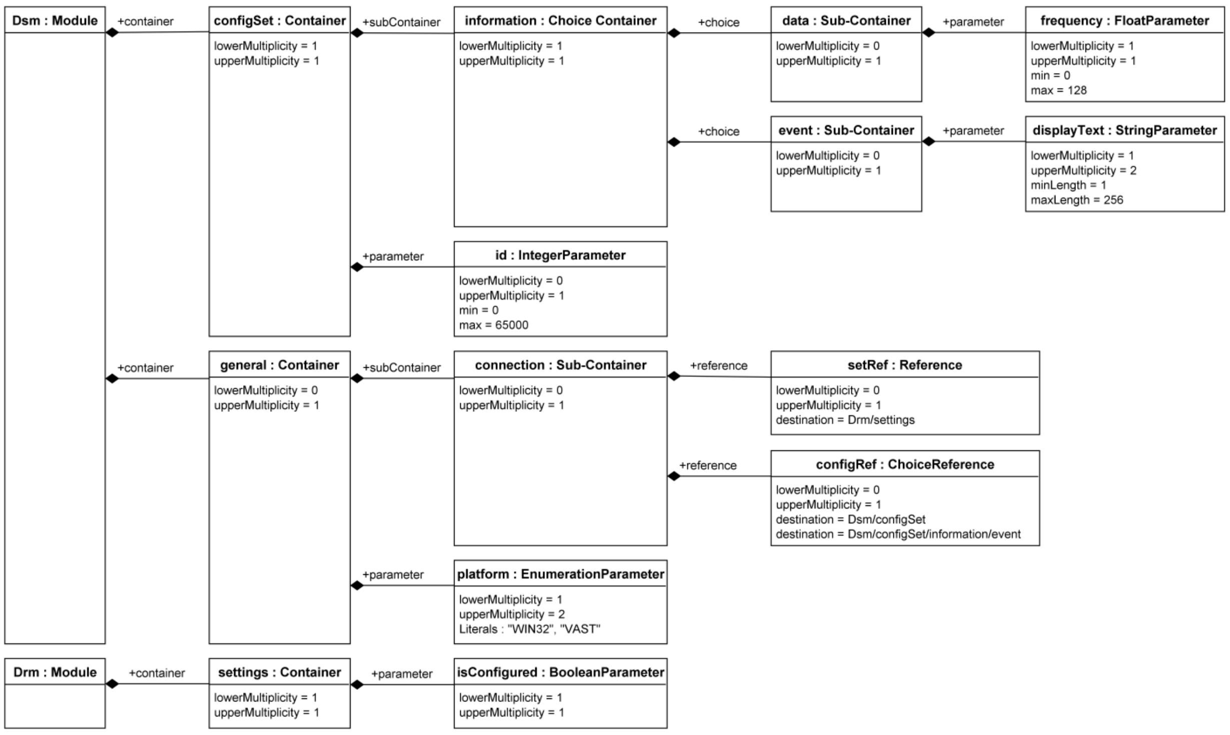


Figure 21 - The structure of the Dsm and the Drm modules.

As described in figure 21 the Dsm module has two containers. One of them, the 'configSet' container, is mandatory, while the other one, called 'general', is optional. The 'configSet' container has one optional integer parameter used to set an id with a value between 0 and 65000. It also has a mandatory choice sub-container that contains more information about the module. The two choices of this 'information' sub-container are called 'data' and 'event' and they are both optional. The 'data' sub-container has a mandatory float parameter used for setting a frequency between 0.0 and 128.0, while the 'event' sub-container has a mandatory String parameter with an upper multiplicity of two and a maximum length of 256. The 'general' container of the Dsm module has a mandatory enumeration parameter used to set the platform with the values ‘WIN32’ or ‘VAST’. The upper multiplicity of this ‘platform’ parameter is two. The ‘general’ container also has an optional sub-container called ‘connection’ which has two optional references. The first one, ‘setRef’, is used to reference a container from the Drm module. The second one is a choice reference called ‘configRef’ and it is an internal reference since it is used to connect to the ‘configSet’ container of the Dsm module or to the ‘event’ sub-container from the same module. The secondary module used in this example, called ‘Drm’ has a mandatory container ‘settings’ which includes a mandatory boolean parameter called ‘isConfigured‘.

5.2. Random Generator Algorithm

The random generator creates one configuration at a time for each one of the primary modules given as an input. The algorithm we used in its implementation is described using an activity diagram in figure 22. It uses a pseudorandom-number generator from java.util.Random for choosing between multiplicities and values for each element. This pseudorandom-number generator was deemed sufficiently random for our purpose. More detailed activity diagrams are included for the main steps. The algorithm is explained in this section using the example described in section 5.1.

The algorithm includes steps for processing all the elements of the primary modules, such as containers, sub-containers, references and parameters. After all the elements are processed, values are added to the references and a List of Configuration Elements is passed to the Configuration Creator.

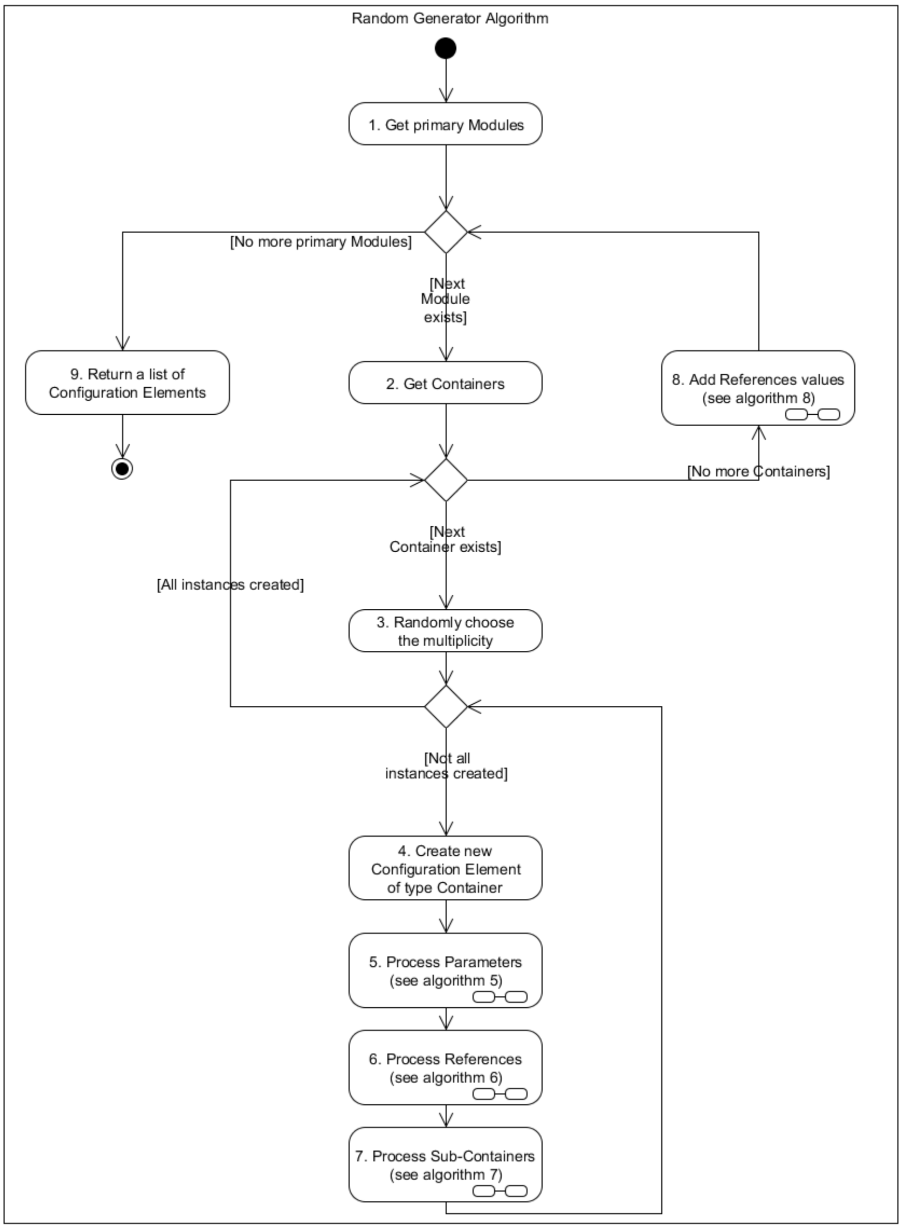

Random Generator Algorithm
1. Get primary Modules
[No more primary Modules]
[Next Module exists]
9. Return a list of Configuration Elements
2. Get Containers
8. Add References values (see algorithm 8)
[No more Containers]
[Next Container exists]
[All instances created]
3. Randomly choose the multiplicity
[Not all instances created]
4. Create new Configuration Element of type Container
5. Process Parameters (see algorithm 5)
6. Process References (see algorithm 6)
7. Process Sub-Containers (see algorithm 7)

Figure 22 - Random Generator Algorithm Activity Diagram.

The first step of this algorithm is to get from the input arguments all the primary modules that need to be included in the configuration. In this example, there is only one primary module, so this step results in getting the Dsm module.

In step 2, the generator gets all the containers from the current primary module, which in our case results in getting both containers of the Dsm module, called ‘configSet’ and ‘general’. Next, the steps from 3 to 7 are repeated for each of these containers.

Step 3 is used to randomly choose a multiplicity between the lower and the upper multiplicity for the current container, so in our example a multiplicity is chosen for the ‘configSet’ container. This multiplicity can only be 1 since both the lower and the upper multiplicities are equal to 1.

In step 4, a new Configuration Element is created. As described in the Architecture chapter, a Configuration Element contains its value and its parameter definition information. The parameter definition information consists of the module it belongs to, its short name, type, definition object and other information. When an element needs to be created, the random generator creates one Definition Element which holds the information related to the parameter definition file and one Configuration Element which holds the information related to the configuration. The Configuration Element contains one and only one Definition Element.

In this example the newly created Configuration Element has a value set to ‘On’ and a Definition Element with the type attribute set to ‘container’, a module attribute set to the name of the current module ‘Dsm’ and a name attribute set to ‘configSet_0’. It also contains a definitionObject attribute set to ‘Dsm/configSet’. The definitionObject attribute points to the element in the parameter definition file. The relation between the Configuration Element and the Definition Element is illustrated in figure 23.

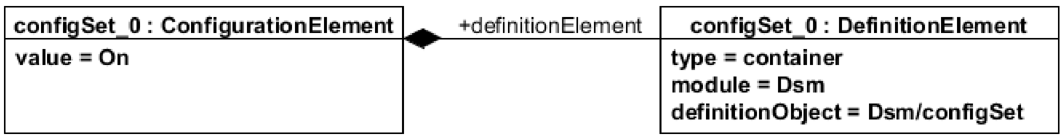

configSet_0 : ConfigurationElement
value = On
+definitionElement
configSet_0 : DefinitionElement
type = container
module = Dsm
definitionObject = Dsm/configSet

Figure 23 - The first Configuration Element and its Definition Element.

For the purpose of readability and a more compact view of the configuration, the Configuration Elements and the Definition Elements are illustrated in the rest of this section as one single element, as described in figure 24. Since all the elements of the parameter definition file have unique names in this example, the short name is sufficient to identify its definition object.

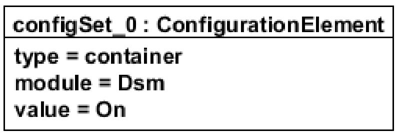


Figure 24 - The first Configuration Element in a more compact view.

The algorithm continues with step 5 where the parameters of the current container are processed. Step 5 is described in figure 25.

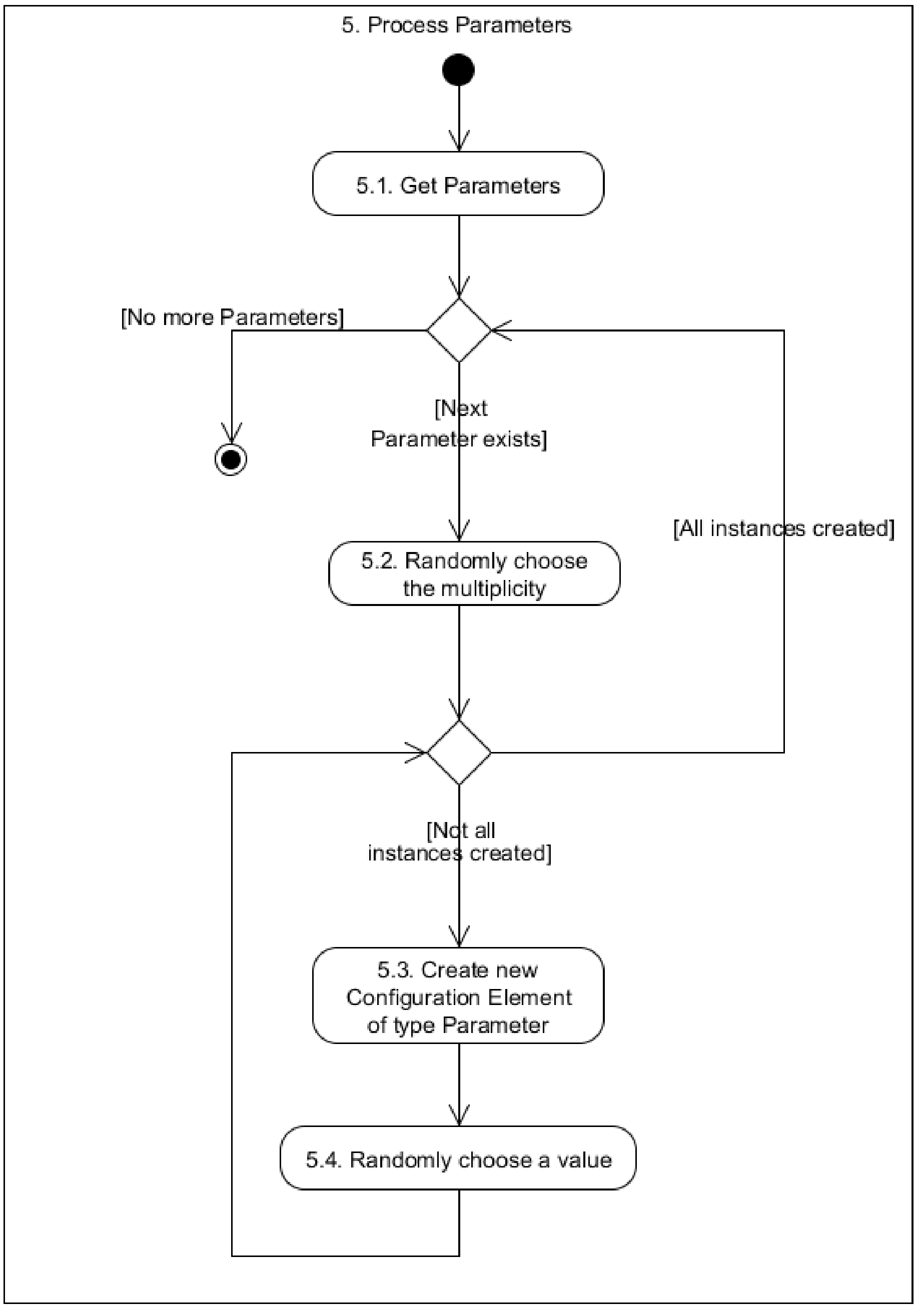

5. Process Parameters
5.1. Get Parameters
[No more Parameters]
[Next Parameter exists]
[All instances created]
5.2. Randomly choose the multiplicity
[Not all instances created]
5.3. Create new Configuration Element of type Parameter
5.4. Randomly choose a value

Figure 25 - Algorithm 5: Process Parameters activity diagram.

When processing parameters, the step 5.1. is getting all parameters of the current container. This time it results in getting the 'id' integer parameter of the 'configSet' container.

In step 5.2. a multiplicity between the upper and the lower multiplicity is randomly chosen. In this case the generator has to choose between 0 and 1. If value 0 is chosen, this parameter is not included in the configuration. In case the multiplicity 1 is chosen, only one instance of this parameter is added to the configuration. We assume that the chosen multiplicity in this step is 1.

The next step is 5.3. where a new Configuration Element of type Parameter is created. This time the generator creates a new Configuration Element with the type attribute set to ‘parameter’ and the name set to ‘id_0’. This new Configuration Element is added as a child of the ‘configSet_0’ Configuration Element and its module attribute is set to ‘Dsm’.

Next, in step 5.4. the generator has to randomly choose and assign a value to the current parameter. The process of choosing this value depends on the parameter type. For a boolean parameter, the generator has to choose a value between 0 (false) and 1 (true). For an integer and a float parameter, the generator randomly picks a value between the minimum and the maximum values defined for these parameters in the parameter definition file. For a parameter of type enumeration, the generator randomly chooses one of the literals defined for the parameter. For a string parameter, a random string is generated that conforms to the regular expression defined for the parameter. This is done by using a java library called Xeger (Xeger, 2009). If this regular expression is not defined, a value equal to the default value or the short name is assigned.

In our example, since the parameter ‘id_0’ is an integer and its value can be between 0 and 65000, we assume that the chosen value in this step is 100. After adding this new Configuration Element and setting a value for it, the configuration’s structure looks like in figure 26.

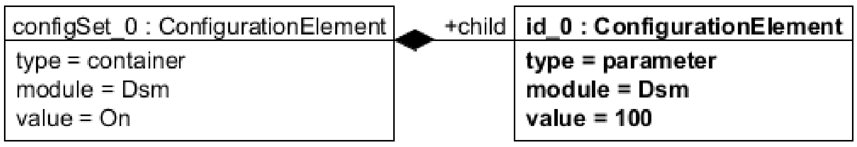


Figure 26 - Configuration Element List structure after step 5.4.

Since no more instances of the ‘id’ parameter can be added to the configuration and there are no more parameters in the ‘configSet’ container, the algorithm will continue with step 6 where the references of the current container are processed. This step is divided into several other steps as described in figure 27.

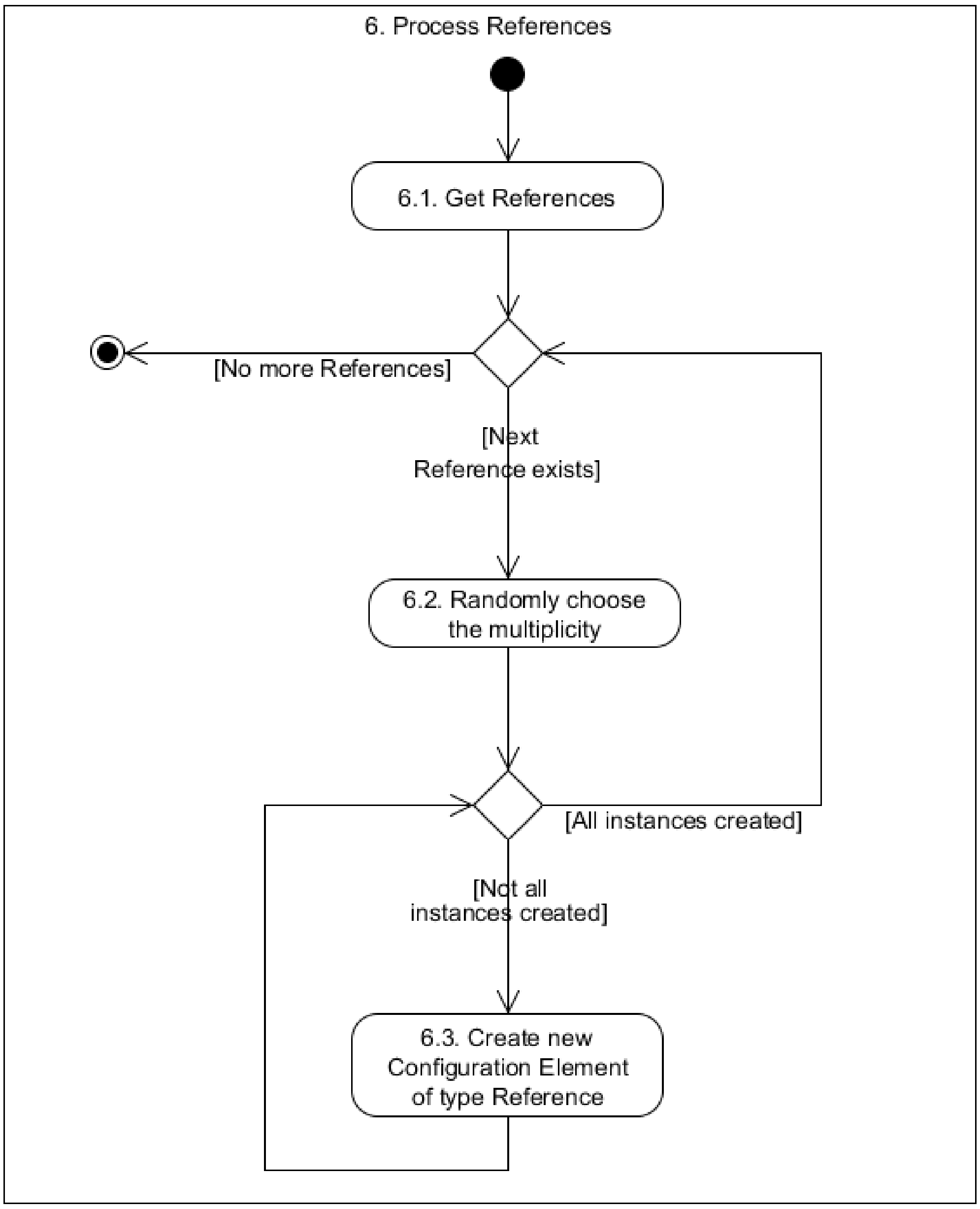

6. Process References
6.1. Get References
[No more References]
[Next Reference exists]
6.2. Randomly choose the multiplicity
[All instances created]
[Not all instances created]
6.3. Create new Configuration Element of type Reference

Figure 27 - Algorithm 6: Process References activity diagram.

When processing references, the first thing the generator does is to get all references of the current container. Since in this example the ‘configSet’ container does not have references, the algorithm does not continue with steps 6.2. and 6.3. where a multiplicity should be chosen for each reference and a new Configuration Element of type Reference should be created. These steps are explained later in this section, when processing the references included in the ‘connection’ sub-container.

The next step in the algorithm is number 7 where all the sub-containers of the current container are processed. This step is divided into several other steps as described in figure 28.

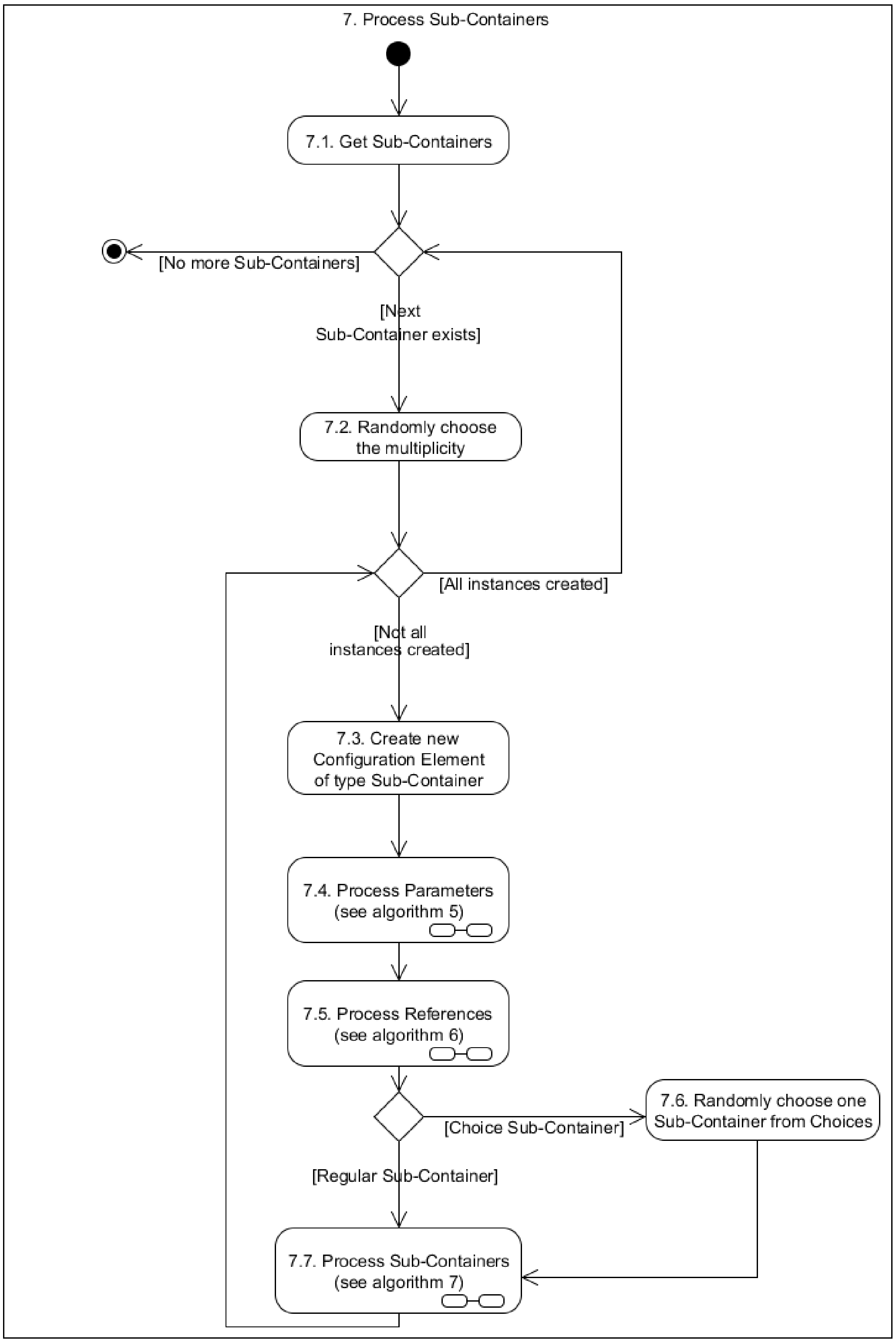

7. Process Sub-Containers
7.1. Get Sub-Containers
[No more Sub-Containers]
[Next Sub-Container exists]
7.2. Randomly choose the multiplicity
[All instances created]
[Not all instances created]
7.3. Create new Configuration Element of type Sub-Container
7.4. Process Parameters (see algorithm 5)
7.5. Process References (see algorithm 6)
[Choice Sub-Container]
7.6. Randomly choose one Sub-Container from Choices
[Regular Sub-Container]
7.7. Process Sub-Containers (see algorithm 7)

Figure 28 - Algorithm 7: Process Sub-Containers activity diagram.

When processing sub-containers, the first step is to get all the sub-containers included in the current container. In our example, this step results in getting the only sub-container, called ‘information’ according to figure 21. In step 7.2. a random multiplicity has to be chosen for the current sub-container, but in our case in can only be equal to 1.

Next, during step 7.3., a new Configuration Element of type sub-container is created. In our case, the new element is a child of the 'configSet_0' Configuration Element. The value is set to 'On', the module to 'Dsm', the type is 'sub-container' and the name given is 'information_0'. The new structure of the Configuration Element List is illustrated in figure 5.8.

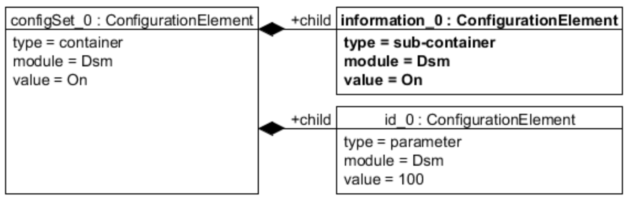


Figure 29 - Configuration Element list structure after step 7.3.

Steps 7.4. and 7.5. are used to process the parameters and the references of the current sub-container. In our example these steps are skipped since our current sub-container 'information' does not have any parameters or references to process.

For a choice sub-container, which is the case for the 'information' sub-container, the algorithm continues with step 7.6. where a choice is randomly chosen to be included in the current sub-container. As described in figure 21, the 'information' sub-container has two choices: a 'data' sub-container and an 'event' sub-container. At this point the algorithm has to randomly choose only one of the two sub-containers to process. We assume that it chooses the 'data' sub-container before continuing to step 7.7. Therefore, this Configuration Element List will not include the 'event' sub-container or its 'displayText' parameter.

Step 7.7. is a recursive call of the step 7, so it is processing the sub-containers included in the current sub-container. In our example the current sub-container is 'information'. Since this part of the algorithm has already been explained in this example, we are just taking a quick look at what is happening in this step. Firstly, the generator gets the sub-containers of the 'information' sub-container. In this case it just gets the sub-container it chose in the previous step, called 'data'. A new Configuration Element is created since once a choice is picked, it has to be added to the configuration indifferent what its lower multiplicity is. The name given to this Configuration Element is 'data_0'. It is a child of the 'information_0' Configuration Element and it includes a child 'frequency_0' with a randomly chosen value of 75.5. The structure of the Configuration Element List is illustrated in figure 30.

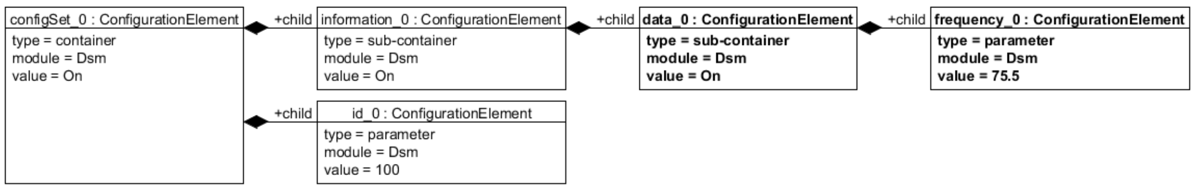


Figure 30 - Configuration Element List structure after step 7.7.

After processing the first container of the Dsm module, the algorithm continues, according to figure 22, with repeating the steps 3-7 for the second container, called 'general'. Step 3 generates a multiplicity between 0 and 1. We assume the value 1 is chosen. Therefore, step 4 creates a new Configuration Element of type 'container' with the name 'general_0'. In step 5 the only parameter of the general container, called 'platform', is processed. This time step 5.2. can choose a multiplicity up to 2. For this example, we assume that value 2 is chosen, so the algorithm will create 2 instances of the same parameter and assign a random value for each one. The name of the first Configuration Element of type ‘parameter’ is ‘platform_0’ and the name of the second one is ‘plaftform_1’. The algorithm continues then with step 6, where no references can be processed, since the ‘general’ container does not have any references. Step 7 processes the only included sub-container, called ‘connection’ and it creates a new Configuration Element of type ‘sub-container’, assuming that its randomly chosen multiplicity is 1. The name of the new Configuration Element is set to ‘connection_0’. Step 7.4. is skipped since this sub-container does not have any parameters and the algorithm continues with step 7.5. where the references are processed. Before continuing with step 7.5. we can define the structure of the list of Configuration Elements as described in figure 31.

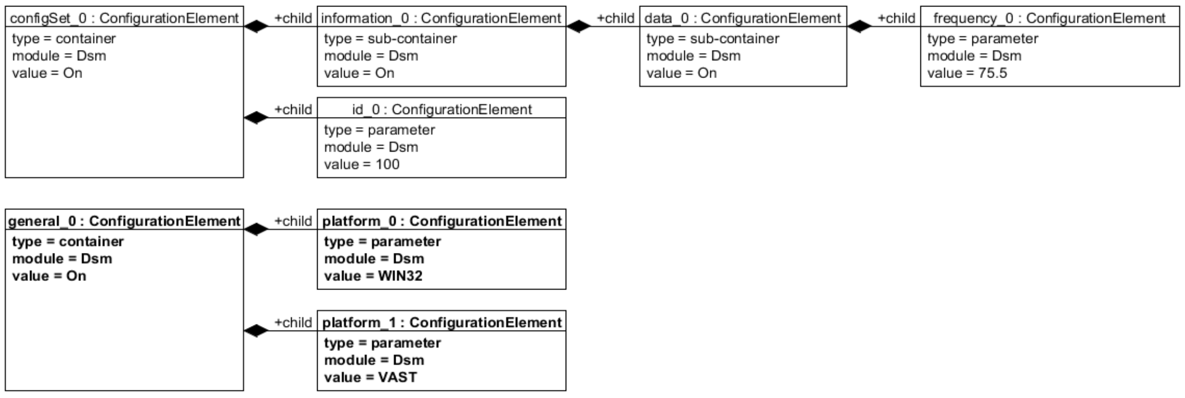


Figure 31 - Configuration Element List structure after step 7.4.

When processing references in step 7.5. the process is the same as in step 6 which is divided into several other steps as described in figure 27. The first sub-step is to get all the references included in the current sub-container. This time there are two optional references contained in ‘connection’ and they are called ‘setRef’ and ‘configRef’. The next sub-steps included in this process are repeated for each one of the two references.

First time, the generator has to choose a random multiplicity for the ‘setRef’ reference and we assume that it is equal to 1. Next, a new Configuration Element of type reference needs to be created. The new Configuration Elements added it named ’setRef_0’. The values for the Configuration Elements of type reference are added later, in step 8, after all elements have been processed.

The second time, the generator has to choose a multiplicity for the ‘configRef’ reference. In this case it chooses a random multiplicity which we assume it is equal to 1 and creates a new Configuration Element of type ‘reference’ and name ‘configRef_0’. The list of Configuration Elements has now the structure defined in figure 32.

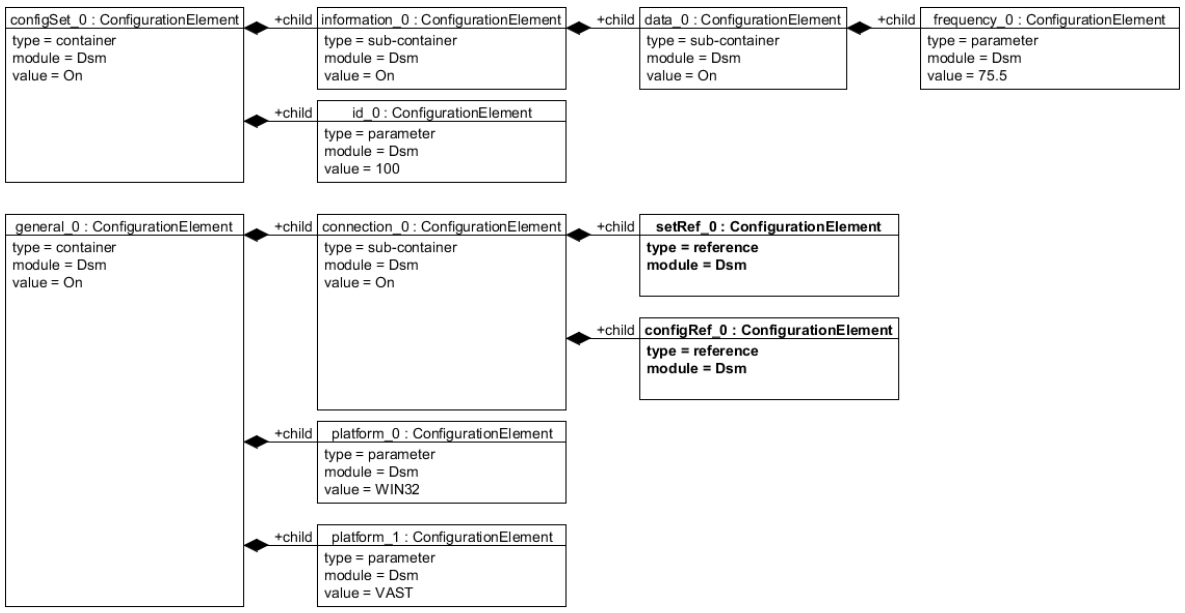


Figure 32 - Configuration Element List structure after step 7.5.

Since the ‘connection’ sub-container is not a choice sub-container, the algorithm continues with step 7.7. where the included sub-containers of the current ‘connection’ sub-container need to be processed. This step is skipped since the ‘connection’ sub-container does not include any sub-containers of its own. Next, since all the elements of the primary module ‘Dsm’ have been processed, the algorithm continues with step 8, where values are being added to the created Configuration Elements of type ‘reference’ as described in figure 33.

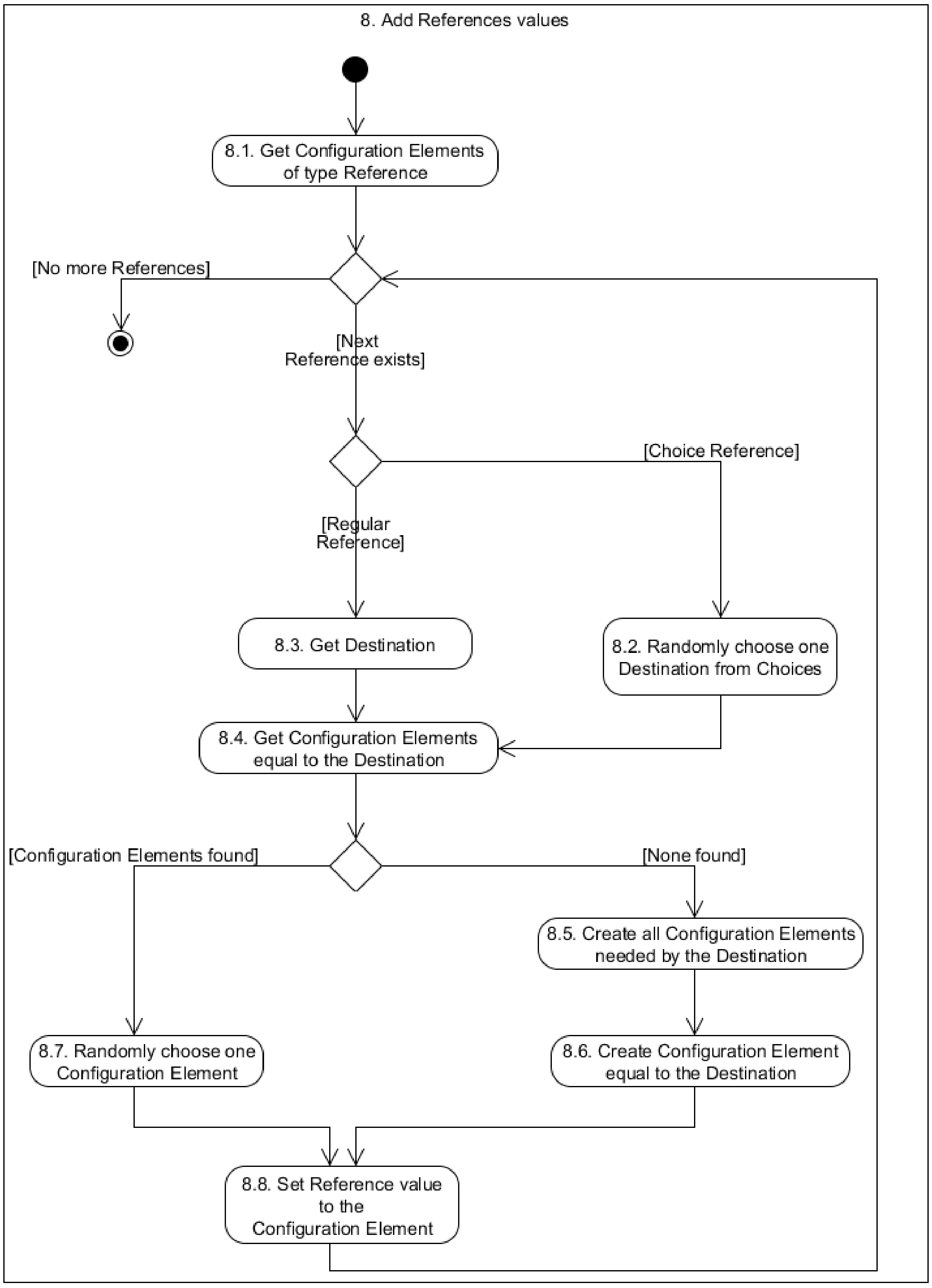

8. Add References values
8.1. Get Configuration Elements of type Reference
[No more References]
[Next Reference exists]
[Choice Reference]
[Regular Reference]
8.3. Get Destination
8.2. Randomly choose one Destination from Choices
8.4. Get Configuration Elements equal to the Destination
[Configuration Elements found]
[None found]
8.5. Create all Configuration Elements needed by the Destination
8.7. Randomly choose one Configuration Element
8.6. Create Configuration Element equal to the Destination
8.8. Set Reference value to the Configuration Element

Figure 33 - Algorithm 8: Add references values activity diagram.

When adding values to the Configuration Elements of type ‘reference’, the step 8.1. is to get all these elements, so basically the algorithm loops over all Configuration Elements created and finds the ones that are of type ‘reference’. According to figure 32, there are two Configuration Elements of this type defined so far ‘setRef_0’ and ‘configRef_0’. The steps from 8.2. to 8.8. are repeated for each one of these elements.

First time, since ‘setRef_0’ is not defined as a choice reference, the algorithm continues with step 8.3. where it gets the destination of the current reference element. In this example, this is an external destination pointing to the ‘settings’ container in the Drm module.

Next, in step 8.4., the generator tries to find an existing Configuration Element equal to the destination. In our case, it looks for a Configuration Element that is defined by the ‘settings’ container of the Drm module. Since we have not processed the Drm module yet, this Configuration Element is not found, therefore the algorithm continues with step 8.5.

In step 8.5. all the Configuration Elements that need to exist in order for the destination to be created, i.e. the entire parent hierarchy, are added to the configuration. In this example, the destination is the ‘settings’ container, which is not a child of another container, so the generator does not need to create any other Configuration Elements, i.e. parents. This step is skipped and the algorithm continues with step 8.6.

In step 8.6. a new Configuration Element equal to the destination needs to be created. In our case this step results in creating a new Configuration Element of type ‘container’, called ‘settings_0’. The algorithm also processes its parameters and creates another Configuration Element of type ‘parameter’, called ‘isConfigured_0’.

Since the new Configuration Elements are not part of the ‘Dsm’ module, but of the ‘Drm’ module, their module attribute is set to ‘Drm’, i.e. a new module has been added to the configuration. After all attributes are set for the two new Configuration Elements, the algorithm continues with step 8.8. where a value is set for the Configuration Element of type ‘reference’ pointing to the newly created Configuration Element. In this example, the value of the ‘setRef_0’ Configuration Element is set to refer to ‘settings_0’. Following this step, the Configuration Element List has the structure as described in figure 34.

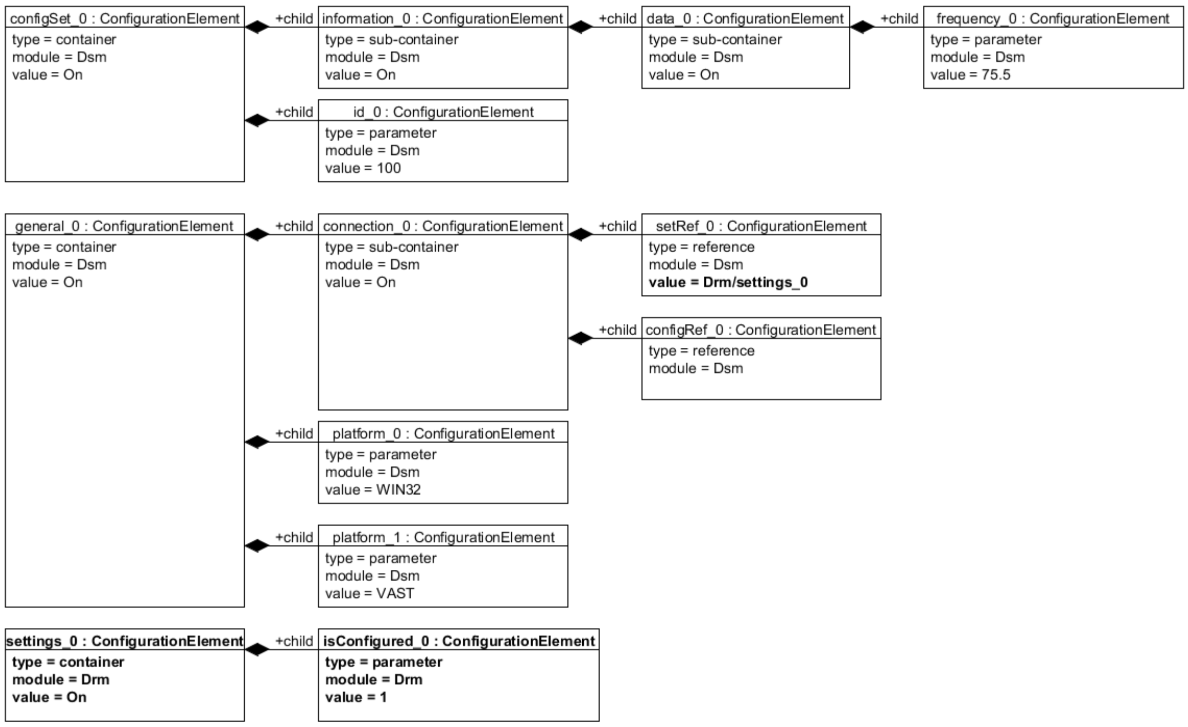


Figure 34 - Configuration Element List after step 8.8.

The second time the algorithm comes to step 8.2. it is processing the second Configuration Element of type reference called ‘configRef_0’. Since this element is defined by a choice reference, in step 8.3. one of the two possible destinations is chosen. According to figure 21, the two possible destinations are ‘Dsm/configSet’ and ‘Dsm/configSet/information/event’. We assume that the chosen destination in this step is ‘Dsm/configSet’.

Next, in step 8.4., the algorithm searches through all the Configuration Elements created so far and finds only one that matches the selected destination. This is the ‘configSet_0’ Configuration Element which has its module attribute set to ‘Dsm’.

Because a matching element was found, the algorithm continues with step 8.7. where it randomly chooses between all elements found. Having only one such element, the algorithm chooses it and continues with step 8.8. where it assigns the value to the ‘configRef_0’ Configuration Element. The structure of the final Configuration Element List is described in figure 35.

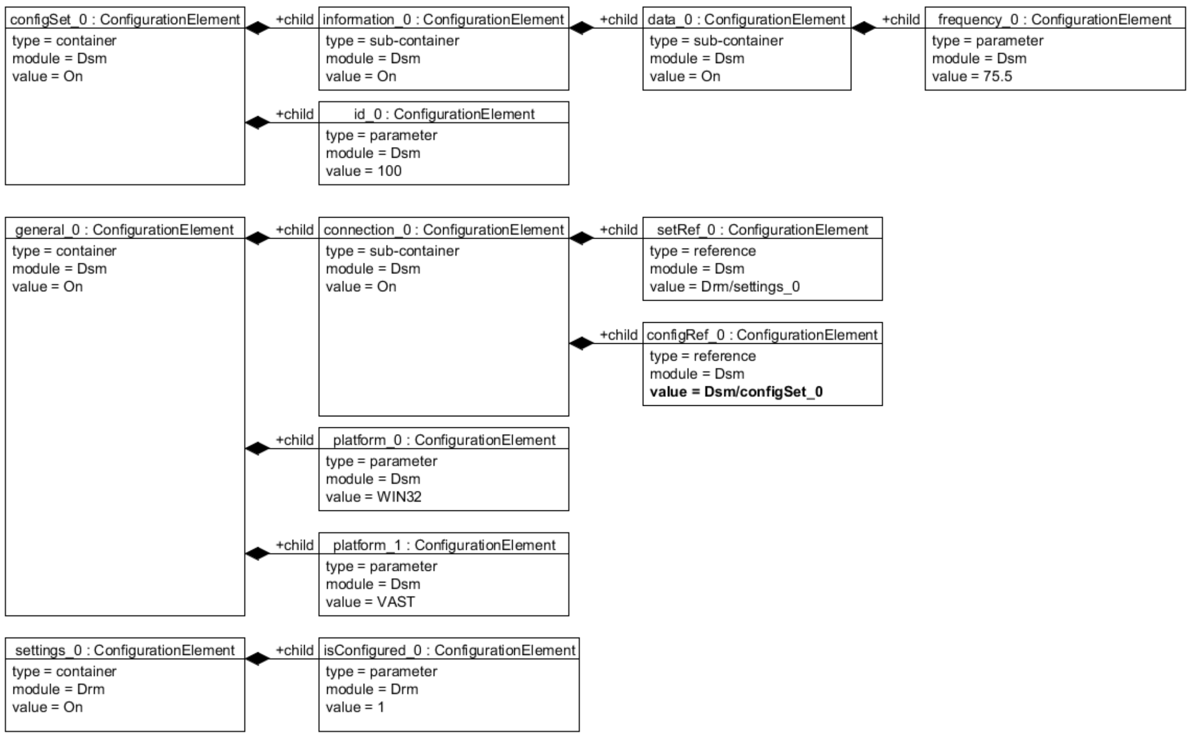


Figure 35 - Configuration Element List structure after step 8:

Now that the random generator has created a Configuration Element List, it has to pass it to the Configuration Creator, which creates the actual configuration in the format of an ARXML file.

5.3. Pairwise Generator Algorithm

This section describes the pairwise algorithm, first as an activity diagram with text explaining the diagram. It then goes on to explain restrictions that must be handled by the pairwise algorithm and use of optimisations within the algorithm. The section ends by reinforcing and clarifying the initial details by way of example.

The example will start by building the Definition Element Table which is a data structure representing the content of the parameter definition. The contents of the Definition Element Table are used for creating all pairs. The pairs are identified by pairing up all values of a given element in the table with all values of the preceding elements in the Definition Element Table. For example if the second element is being processed, all pairs between the values of the first and second element are calculated. If the third element is being processed, all pairings are computed between the values of the third element and the values of the first element and between the values of the third element and the values of the second element. This all pairs calculation is conducted for all elements in the Definition Element Table so that all existing pair combinations are addressed.

When the algorithm computes all pairs for a given element it then adds them to the Configuration Element Table. This is a data structure and each row is an internal representation of an AUTOSAR configuration.

There are two main strategies for adding pairs to the table. These are adding vertically and adding horizontally. Given a Configuration Element Table and a pair that needs to be added to the table, if the pair cannot be added to any of the existing rows because there are already elements of the same type occupying all of the rows then the algorithm needs to add vertically to satisfy the pair. This means that a new row will be added and the unsatisfied pair will be added to this row. The addition of this new row is equivalent to adding a new AUTOSAR configuration.

When the algorithm is finished processing an element from the Definition Element Table it then moves horizontally to the next element i.e. it selects the next element from the Definition Element Table and ensures that the pairs of its values and the previous elements' values are covered in the Configuration Element Table.

The algorithm continues its processing until all identified pairs have been represented in at least one of the rows of the Configuration Element Table. As a result all rows are to be considered together to be complete with respect to covering all possible pair combinations.

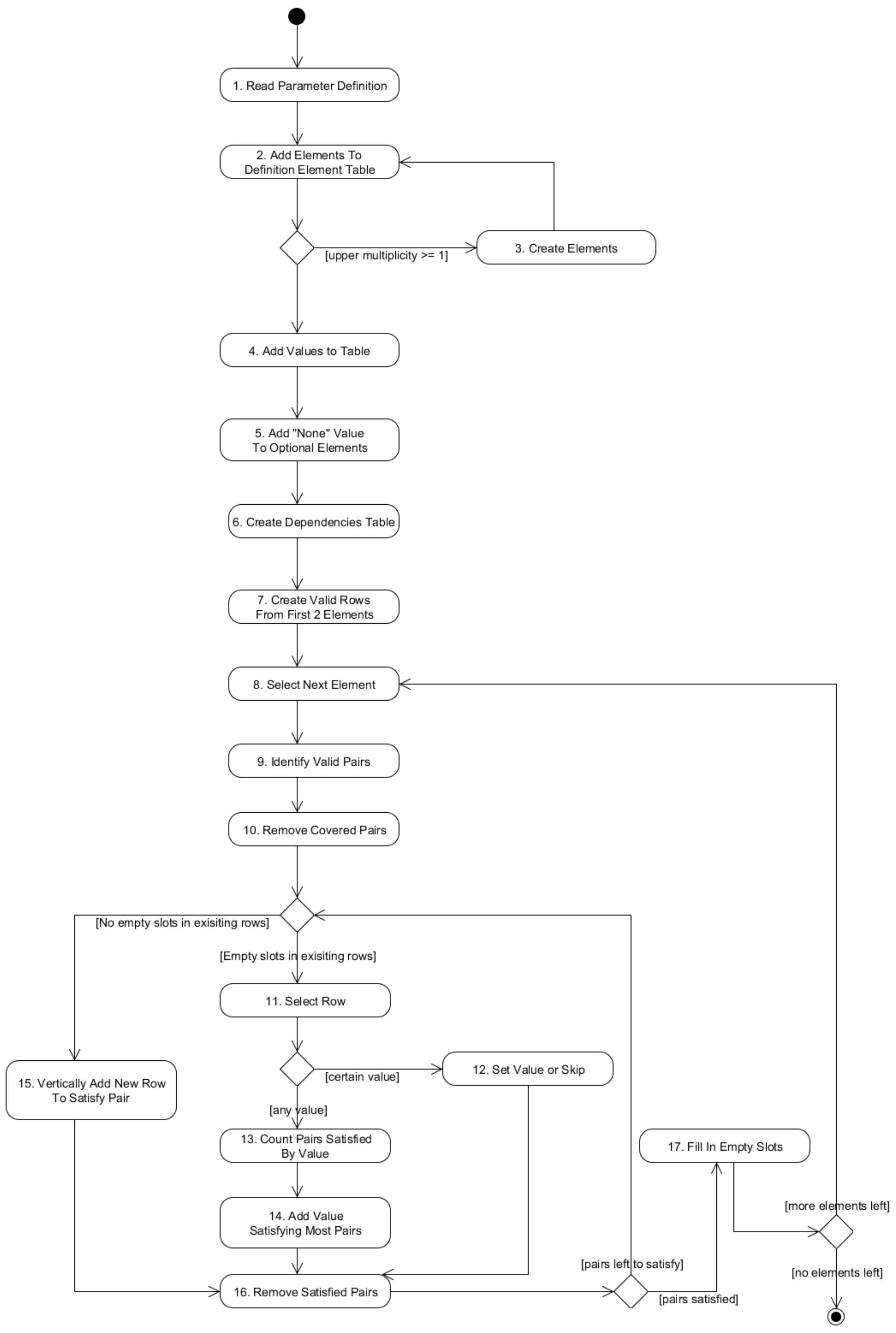
5.3.1 Pairwise Algorithm Description

The following activity diagram in figure 36 shows the process that the pairwise algorithm follows in processing all the pairs. First it builds the Definition Element Table. All elements from the parameter definition are converted into Definition Elements and added to the table. If an element has an upper multiplicity greater than or equal to 1, then the corresponding number of instances of that element will be added to the Definition Element Table. If an element has a lower multiplicity of 0 then a value called None will be added for all instances of that element in the Definition Element Table. A value of None means that this element instance will not exist in the final AUTOSAR configuration. If the lower multiplicity is greater than 0, for example 2, then all instances of that element above the lower multiplicity will also have a value of None whereas the first two instances will not. Along with the None value, the elements' values from the parameter definition will be added.

When the Definition Element Table is complete, the elements of the first two elements are paired up and for each valid pairing a new row is added to the Configuration Element Table. More detail of what constitutes an invalid pair follows after this description of the activity diagram.

When rows have been created satisfying all pairs for the first two elements the next element is read from the Definition Element Table. All pairs that are not covered in the Configuration Element Table between the values of the previous elements and the values of the element being processed are identified. If there is an empty slot for the new element in an existing row in the Configuration Element Table then the pairwise algorithm will find which valid value for the new element will cover most uncovered pairs if added to the row. If there are no empty slots then a new row is added that will satisfy an uncovered pair. It is possible to add multiple uncovered pairs to the same row if they are paired up with different elements and they are valid within the row.

These steps are repeated until all the pairs have been added to the Configuration Element Table. Any empty slots from the first column up to the current column are assigned a valid value for completion. Then the next element is selected and the process repeated until all elements from the Definition Element Table have been processed and the pairings of all their values are represented within the Configuration Element Table.



1. Read Parameter Definition
2. Add Elements To Definition Element Table
[upper multiplicity >= 1]
3. Create Elements
4. Add Values to Table
5. Add "None" Value To Optional Elements
6. Create Dependencies Table
7. Create Valid Rows From First 2 Elements
8. Select Next Element
9. Identify Valid Pairs
10. Remove Covered Pairs
[No empty slots in exisiting rows]
[Empty slots in exisiting rows]
11. Select Row
[certain value]
12. Set Value or Skip
[any value]
15. Vertically Add New Row To Satisfy Pair
13. Count Pairs Satisfied By Value
14. Add Value Satisfying Most Pairs
16. Remove Satisfied Pairs
[pairs left to satisfy]
[pairs satisfied]
17. Fill In Empty Slots
[more elements left]
[no elements left]

Figure 36 - Pairwise Generator Algorithm.

Invalid Pairs and Values

When describing the activity diagram in figure 36, the concept of valid pairs was introduced. In its simplest form the pairwise algorithm pairs up all the values of the elements it is provided and builds a table to cover all of these pairs. As our case study was conducted using AUTOSAR parameter definitions there were certain pairings that become invalid due to the implicit dependencies between the elements of the parameter definition. These invalid pairs can be split into two types, those that are **invalid at all times** and will not be covered by the pairwise algorithm and those that are **invalid in a given row** due to values that already exist in the row but must be covered by pairwise. For the second type it can be just a single individual value that the algorithm is trying to add that causes the row to become invalid.

The following lists the cases where pairs are invalid at all times.

1. Any pairings where a value that is not None is provided to a Configuration Element but its parent or any element in its parent hierarchy has a value of None. For example if the pair contains a parameter called 'frequency' with a value of 60.0 but its parent 'data' has a value of None then this parent will not exist in the final AUTOSAR configuration. The 'frequency' parameter also cannot exist in this case so pairs such as these are removed.

2. Pairs between a choice container and values within the choice container. The value of the choice container has an impact on the value of the choices so they are considered one element by pairwise and not something to process pairs for. For example, if we have a choice container called 'information' that has chosen the 'data' sub-container then the 'data' sub-container must have a value of On. The opposite is also true, if the 'data' sub-container has a value of On then the 'information' choice container must have a value of data. Their pairings do not need to be considered as they have this direct relationship.

3. If both elements of a pair are siblings within a choice then their pairs are removed as the siblings cannot co-exist within a choice. Pairs where one of the siblings has a value of None are covered implicitly as a sibling must have a value of None if its sibling has a value that is not None.

4. If a reference has a value referencing a container but it is being paired with a container that is in the parent hierarchy of the referenced container and has a value that is None then this pair is invalid. For example if a reference called 'childRef' is referencing a container called 'childContainer' but it is being paired with an element called 'parentContainer' where 'parentContainer' is the parent of 'childContainer' and the value of 'parentContainer' is None then this pairing is not valid.

5. If a reference refers to an element within a choice but it is being paired with another container within the same choice container, but in a different choice, then this pair is invalid. For example if we have a choice container with two choices 'choice1' and 'choice2' and the pair includes a reference referring to a parameter in 'choice1' but the other side of the pair gives 'choice2' a value of On then this pair is invalid as it would result in both choices being added to the configuration when only one choice can be added.

6. If the pair contains a reference that does not have a value of None and the container being referenced, but the container being referenced has a value of None then this is an invalid pair. For example, if one side of the pair is a reference called 'containerRef' which references a container called 'container1' and the other side of the pair is

‘container1’ with a value of None then this pair is invalid. In this case the resulting configuration would have a reference to a container that does not exist.

7. If there is an optional container referenced by a mandatory reference then if there is only one instance of the optional container the value of the container cannot be None. If there are only two instances a pair with both instances set to None is invalid.

8. If there are any pairings where a reference has been given a value other than None and it references a choice or any element within the choice when the choice has not been chosen, and the choice container is the other element in the pair then this pair is invalid. For example if one side of the pair is a reference to a choice called ‘choice1’ but the other side of the pair is the choice container with a value of ‘choice2’ then the pair is invalid. ‘choice1’ and ‘choice2’ cannot co-exist in the configuration.

The second type of invalid pairs/values are those that are invalid within a row due to the values that have already been added to the row. The following list contains these type of restrictions when adding pairs/values.

1. If the row contains a parent or container within the parent hierarchy that is set to None then the child value can only be set to None.

2. If attempting to add an element that exists within a choice but another choice was chosen by the choice container then this element can only be set to None.

3. If trying to reference an element within a choice but the choice container has made a different choice, then the reference cannot refer to this element.

4. If referencing an element then the referenced element must exist.

5. When adding a value for an element that is not None, the parent hierarchy for the element must also be added. If the element is a reference then the referenced element also needs to be added along with the parent hierarchy of the referenced element.

6. When adding a referenced container and a reference to your type of container is mandatory and you are the last of your type to be assigned a value and the reference has not yet been created then your container must be added to the row along with the reference.

7. If the reference is a symbolic name reference then when the referenced element is added the symbolic name parameter it contains must also be added. More detail about this type of reference can be seen in section 2.2.3.

8. A container must be given a value of On if the row contains a choice container that has chosen it.

9. If adding a reference that is a mandatory reference but within an optional parent hierarchy then if the value of the reference is None its parent hierarchy must also be None all the way up the hierarchy up to and including the optional parent. This is is to avoid the invalid state where the parent is On but the mandatory reference has been given a value of None and so will not exist in the resulting AUTOSAR configuration.

10. When adding the parent for a container if the parent is a choice container it must choose the container being added to the row.

These restrictions introduced by the parameter definition dependencies have an impact on the performance of the pairwise algorithm. Firstly it requires additional processing when creating the pairs to add and when adding values to rows in the Configuration Element Table to ensure that invalid elements are not added.

Secondly the addition of one value to a row in the Configuration Element Table can result in adding many other values, for example if a child is added its entire parent hierarchy must be added so the pairwise algorithm loses the ability to choose what values to select in these cases.

Optimizations were added to the pairwise algorithm to counteract the impact of the restrictions. The following are the optimizations that were implemented. Firstly, when processing all pairs between elements there are no dependencies between different elements in the Definition Element Table so it is possible to carry out this processing on multiple cores. We had access to dual cores, so we split this processing over both cores.

Next, when references are looking for the element they reference instead of looking through the entire Definition Element Table a smaller table is used that just contains containers and sub-containers.

Finally, unnecessary looping over the Definition Element Table is reduced. For example there are restrictions that need to know about mandatory references. The list of mandatory references is built once and reused throughout the algorithm.

5.3.2 Pairwise Example

This example uses the modules from figure 21 which were also used for explaining the random algorithm. The numbered steps in the activity diagram in figure 36 are referenced throughout the example. The pairwise algorithm first reads the parameter definition file and creates a Definition Element Table (Steps 1 and 2). In this case it will read the Dsm parameter definition. The Drm parameter definition is used for external references and as the external containers are not part of pairwise then the pairwise algorithm does not make use of this parameter definition. The Definition Element Table contains the elements short name, its values, the type of the element and the relationships between the elements. The ‘id’ element in table 1 has a lower multiplicity of 0 so it is given a value of None along with its values from the parameter definition (Steps 4 and 5). This is to cover pairs where this element is not available. The ‘id’ element is an integer element with a value range from 0 - 65000. As testing for all values would create a huge number of pairings to consider it was discussed with Mecel AB that a limitation of taking the min, the max and randomly choosing a number between the min and the max is an acceptable limitation.

If the element has an upper multiplicity greater than 1 then multiple instances of that element are added to the Definition Element Table to cover pairings with all multiplicities (Step 3). This can be seen in the ‘displayText1’ and ‘displayText2’ elements below. ‘displayText2’ is also given a value of None as the lower multiplicity of displayText is 1 so the second instance is not mandatory. Containers are given a value of On which means that the container is available in the configuration.

References are added to the start of the table. This is to ensure that if a reference is to be included in a configuration then its parents and referenced element must also be included. If there was a mandatory reference to an optional container then one instance of that container must be available. The pairwise algorithm also caters for the addition of containers before references. It needs to do this when adding a new row to satisfy a pair. The pair may not contain references but the mandatory references associated with the pair need to be added to the row of the Configuration Element Table.

The pairwise algorithm processes all internal references in one manner and all external references in another. For internal references the values are a list of all instances of the container(s) they reference. In the example the shortname of the containers are provided as the values. In reality a unique id is also used to ensure the correct containers are being referenced. This detail has been omitted from the example to avoid clutter. The ‘configRef’ element below is an internal reference.

External references contain the value On to indicate that the external reference should be included. The pairwise technique is applied to the primary modules and not to the secondary modules. As we are not applying the pairwise technique to the external parameter definition it is enough for the pairwise algorithm to say that the external reference is On or None if the reference is optional. The referenced external container and its hierarchy can be added after the pairwise processing has completed. The element ‘setRef’ is an example of an external reference.

The Definition Element Table for the example described in section 5.1. is as follows.

setRef	**configRef**	**configSet**	**id**	**information**	**data**	**frequency**	**event**	**displayText1**	**displayText2**	**general**	**platform1**	**platform2**	**connection**
None	None	On	None	data	None	0.0	None	text1	None	None	WIN32	None	None
On	configSet		0	event	On	128.0	On		text2	Default	VAST	WIN32	On
	event		65000			54.0						VAST	
			15000										

Table 1 - Definition Element Table.

Although it cannot be seen in the table above, the Definition Element data structure contains information about parent child relationships. It also maintains information about references and their referenced elements and about choices. This information is required by the pairwise algorithm in order to build valid sets of Configuration Elements. As the example progresses we will explore these relationships in more detail.

Once the Definition Element Table has been constructed, step 7 begins and the pairwise algorithm then takes the first two elements and creates all pair combinations between the values of those elements. Table 2 contains all pair combinations between the first two elements ‘setRef’ and ‘configRef’. Both of these elements are references.

setRef	configRef
None	None
None	configSet
None	event
On	None
On	configSet
On	event

Table 2 - All Pairs for setRef and configRef.

The elements being paired in table 2 have no relationship to each other so all of the pairs are valid. If for example ‘setRef’ was a parent of ‘configRef’ then the pairing (None, event) would be an example of an invalid pair as it is not possible to have a value for a child when the parent is None. As this is not the case, pairwise can start to build the Configuration Element Table covering all of these pairs. As there are 6 new pairs and no rows have been added to the Configuration Element Table, the pairwise algorithm will add the pairs vertically creating 6 new rows.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None												
2	None	configSet	On								On			On
3	None	event	On		event			On			On			On
4	On	None									On			On
5	On	configSet	On								On			On
6	On	event	On		event			On			On			On

Table 3 - Configuration Element Table with first two elements paired.

In the first row of table 3, ‘setRef’ and ‘configRef’ are both None so a row has been added to cover this pair. No other element in this row is set as the references do not exist. In the second row ‘configRef’ has a value of ‘configSet’. This means that ‘configRef’ is referencing the container ‘configSet’. This referenced container needs to exist so ‘configSet’ has been given a value of On. The parent of ‘configRef’ needs to exist in order for ‘configRef’ to exist so its parent, ‘connection’, has been given a value of On. In turn the parent of ‘connection’ also needs to exist. So ‘general’ has been given a value of On. The addition of these extra elements satisfies the relationships and dependencies implicit in the parameter definition. Rows 3 to 6 have been built in the same manner. In row 3 ‘configRef’ references the ‘event’ container. The ‘event’ container is one possible choice within the ‘information’ container so you can see that ‘information’ has been given a value of ‘event’. As ‘information’ is a choice container the choice must match with the referenced element. The empty spaces in the table are configuration values that have not been set yet. ‘setRef’ in row 4 is an external reference. As pairwise does not need to process the externally referenced container, ‘setRef’ has been given a value of On as have its parents ‘general’ and ‘connection’.

In step 8 the algorithm takes the third element, 'configSet', from the Definition Element table, table 1. In step 9 it creates all valid pair combinations between the values of 'configSet' and 'setRef' and the values of 'configSet' and 'configRef'. The resulting all valid pairs list is contained in table 4.

setRef	configSet	configRef	configSet
None	On	None	On
On	On	configSet	On
		event	On

Table 4 - All valid pairings between setRef and configSet and configRef and configSet.

It may be the case that some of these pairs are already covered in the Configuration Element Table, table 3, when adding the first set of pairs. The algorithm begins step 10 by checking for any covered pairs resulting in the list of covered pairs in table 5.

setRef	configSet	configRef	configSet
None	On	configSet	On
On	On	event	On

Table 5 - List of already covered pairs for configSet and setRef and for configRef and configSet.

The algorithm then removes the covered pairs from the list of all possible pairs to establish a list of uncovered pairs. Table 6 contains the list of uncovered pairs.

setRef	configSet	configRef	configSet
		None	On

Table 6 - List of uncovered pairs for configSet and setRef and configRef and configSet.

In this case there is only one pair that is not currently covered in the Configuration Element Table. The algorithm now looks to see if this pair can be added to one of the existing rows that contains empty slots. In table 3, the first row contains the value None for 'configRef' and an empty slot for 'configSet'. Based on steps 11-14 and 16 the algorithm will put the value On into this 'configSet' empty slot and satisfy the pair. Steps 12-14 are trivial at this point as there is only one pair to satisfy. We will see these steps in more detail later in the example. As there are no pairs left to satisfy, any empty slots for the first three columns will be assigned a valid value in step 17. At this stage all pairs have been covered so it does not matter what values these slots are given provided that they do not violate any constraints. An interesting future optimisation could be to leave these slots empty in order to satisfy pairings with elements that are encountered later in the Definition Element Table. Table 7 contains the updated Configuration Element Table with the newly added elements written in bold text.

Case #	**setRef**	**configRef**	**configSet**	**id**	**information**	**data**	**frequency**	**event**	**displayText1**	**displayText2**	**general**	**platform1**	**platform2**	**connection**
1	None	None	**On**											
2	None	configSet	On								On			On
3	None	event	On		event			On			On			On
4	On	None	**On**								On			On
5	On	configSet	On								On			On
6	On	event	On		event			On			On			On

Table 7 - Configuration Element Table after processing configSet.

The algorithm now returns to step 8 and selects the next element moving horizontally. As part of step 9 it creates all valid pair combinations between the values of the ‘id’ element and the values of ‘setRef’, ‘configRef’ and ‘configSet’. The resulting set of all valid pairs are shown in table 8.

setRef	**id**	**configRef**	**id**	**configSet**	**id**
None	None	None	None	On	None
None	0	None	0	On	0
None	65000	None	65000	On	65000
None	15000	None	15000	On	15000
On	None	configSet	None		
On	0	configSet	0		
On	65000	configSet	65000		
On	15000	configSet	15000		
		event	None		
		event	0		
		event	65000		
		event	15000		

Table 8 - All valid pairs for id and setRef, configRef and configSet.

Taking a look at the Configuration Element Table in table 7 we can see that no values have been set for ‘id’ at this point. As a result step 10 completes without removing any existing elements and table 8 above with all the possible pairings is left unchanged. All pairs it contains must be satisfied within the Configuration Element Table. The algorithm first looks for a row with an empty slot for the element it is trying to add. Step 11 selects a row and in this case there is an empty slot in row 1 of table 7 in the ‘id’ column. If the parent of ‘id’ has a value of None then the value of ‘id’ must be set to None. The parent of ‘id’ is ‘configSet’ which in this case has a value of On so the algorithm moves on to step 13 as any value for 'id' is valid. Next, in step 13, the algorithm tries to pick the value it should set 'id' to in row 1 by determining which value will cover the most uncovered pairs from table 8. If the algorithm was to choose None as the value for 'id' then 3 pairs would be satisfied. These are (setRef: None, id: None), (configRef: None, id: None) and (configSet: On, id: None). If the algorithm was to choose 0 then 3 pairs would also be satisfied. These are (setRef: None, id: 0), (configRef: None, id: 0) and (configSet: On, id: 0). Calculating in the same manner for values 65000 and 15000 it can be seen that all values will satisfy three pairings. As such it does not matter what value pairwise selects at this point. Pairwise will always choose the last value when more than one value satisfies the most pairs. As a result the value of 15000 is chosen and added to the table in step 14 resulting in table 9.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	**15000**										
2	None	configSet	On								On			On
3	None	event	On		event			On			On			On
4	On	None									On			On
5	On	configSet	On								On			On
6	On	event	On		event			On			On			On

Table 9 - Configuration Element Table with first set of id pairings satisfied.

The pairs (setRef: None, id: 15000), (configRef: None, id: 15000) and (configSet: On, id: 15000) have been satisfied and are removed from the list of all pairs leaving only unsatisfied pairs in step 16. This list is shown in table 10.

setRef	id	configRef	id	configSet	id
None	None	None	None	On	None
None	0	None	0	On	0
None	65000	None	65000	On	65000
On	None	configSet	None		
On	0	configSet	0		
On	65000	configSet	65000		
On	15000	configSet	15000		
		event	None		
		event	0		
		event	65000		
		event	15000		

Table 10 - Uncovered pairs for id after selecting first value.

The algorithm now moves on to the second row and sees that there is an empty slot under the 'id' column. It again counts the number of pairs that would be satisfied by adding each of 'id's values to the empty slot. If 'id' was set to None then 3 pairs would be satisfied, (setRef: None, id: None), (configRef: configSet, id: None) and (configSet: On, id: None). In a similar manner a value of 0 or 65000 would satisfy 3 pairs. However a value of 15000 would only satisfy one pair, (configRef: configSet, id: 15000). As a result a value of 65000 is chosen. The pairs that are now satisfied are removed from the list of unsatisfied pairs in table 10 (step 16). The value for 'id' is chosen in a similar manner for rows 3 - 6 so the steps will not be repeated in this example. The Configuration Element Table, table 11 contains the selected values for 'id' for the first 6 rows.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000										
2	None	configSet	On	**65000**							On			On
3	None	event	On	**0**	event			On			On			On
4	On	None	On	**None**							On			On
5	On	configSet	On	**15000**							On			On
6	On	event	On	**65000**	event			On			On			On

Table 11 - Configuration Element Table after processing three rows for id.

setRef	id	configRef	id
None	None	None	0
On	0	None	65000
		configSet	None
		configSet	0
		event	None
		event	15000

Table 12 - Table of pairings with id that are not yet covered.

The pairs in the above table are pairs that could not be satisfied in the 6 rows that are currently in the Configuration Element Table. As per step 15 the algorithm will add new rows vertically in order to satisfy any pairs that cannot be added to an existing row. The algorithm takes the first unsatisfied pair. It will check if there is an existing free slot that will satisfy the pair. In this case all rows have a value for 'id' so then the algorithm will continue on and add a new row to satisfy the pairing (setRef: None, id: None) and then remove the pair for the list of uncovered pairs (step 16). The algorithm then takes the next unsatisfied pair (setRef: On, id: 0). This pair cannot be added to any of the 7 rows so an eighth row is added to satisfy this pair and the pair removed from the list of uncovered pairs (step 16). As the value of 'setRef' is On then its parent hierarchy also needs to be turned on so 'connection' and 'general' are also set to On. The resulting Configuration Element Table is shown in table 13.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000										
2	None	configSet	On	65000							On			On
3	None	event	On	0	event			On			On			On
4	On	None	On	None							On			On
5	On	configSet	On	15000							On			On
6	On	event	On	65000	event			On			On			On
7	**None**			**None**										
8	**On**		**On**	**0**							**On**			**On**

Table 13 - Vertical Processing of unsatisfied pairs for id.

The algorithm takes the next unsatisfied pair (configRef: None, id: 0). The algorithm checks for any empty slots that can be used in the existing rows to satisfy this pair. It sees that in the eighth row setting the value of 'configRef' to None would satisfy this pair and so adds the value to this row as per steps 11-16. The next pair (configRef: None, id: 65000) requires a new row (step 15 and 16). (configRef: configSet, id: None) can be satisfied in the seventh row(steps 11-16) and the last three pairs require new rows (step 15 and 16). The resulting Configuration Element Table after processing all of these pairs is provided in table 14.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000										
2	None	configSet	On	65000							On			On
3	None	event	On	0	event			On			On			On
4	On	None	On	None							On			On
5	On	configSet	On	15000							On			On
6	On	event	On	65000	event			On			On			On
7	None	**configSet**	**On**	None							**On**			**On**
8	On	**None**	On	0							On			On
9		**None**	**On**	**65000**										
10		**configSet**	**On**	**0**							**On**			**On**
11		**event**	**On**	**None**	**event**			**On**			**On**			**On**
12		**event**	**On**	**15000**	**event**			**On**			**On**			**On**

Table 14 - Vertical Processing of covering all unsatisfied pairs for id.

Next the algorithm fills in the empty slots up until the ‘id’ column (step 17). The values chosen are unimportant as long as they don’t violate any constraints. The Configuration Element Table after filling in the empty slots is provided in table 15.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000										
2	None	configSet	On	65000							On			On
3	None	event	On	0	event			On			On			On
4	On	None	On	None							On			On
5	On	configSet	On	15000							On			On
6	On	event	On	65000	event			On			On			On
7	None	configSet	On	None							On			On
8	On	None	On	0							On			On
9	**None**	None	On	65000										
10	**None**	configSet	On	0							On			On
11	**None**	event	On	None	event			On			On			On
12	**None**	event	On	15000	event			On			On			On

Table 15 - Configuration Element Table after processing the id element.

The next element to process is the ‘information’ element (step 8). The pairwise algorithm considers all valid pairings with ‘information’ as we have seen previously (step 9). The result is table 13.

setRef	information	configRef	information	configSet	information	id	information
None	data	None	data	On	data	None	data
None	event	None	event	On	event	None	event
On	data	configSet	data			0	data
On	event	configSet	event			0	event
		event	event			65000	data
						65000	event
						15000	data
						15000	event

Table 16 - All valid pairs with the information element.

It can be seen in table 16 that there is no pair (configRef: event, information: data). This is because this is an invalid pair and is removed by the pairwise algorithm. The pairing is invalid because ‘event’ and ‘data’ are both contained within the same ‘information’ choice container and so it is not possible for ‘configRef’ to reference ‘event’ when the ‘information’ choice container has chosen ‘data’. The pairwise algorithm removes the pairs that have already been covered in the Configuration Element Table, table 15 (step 10). As a result of this the pairs identified are reduced to the set of pairs in table 17 which will then be processed in the same manner (steps 11-17) as previous elements so that they become covered in the Configuration Element Table, table 18. As the 'information' element is a child of 'configSet' then 'configSet' must have a value of 'On' before 'information' can be assigned a value other than None. This parent-child restriction holds true for all elements.

setRef	**information**	**configRef**	**information**	**configSet**	**information**	**id**	**information**
None	data	None	data	On	data	None	data
On	data	None	event			0	data
		configSet	data			65000	data
		configSet	event			15000	data

Table 17 - All uncovered pairs for the information element.

Case #	**setRef**	**configRef**	**configSet**	**id**	**information**	**data**	**frequency**	**event**	**displayText1**	**displayText2**	**general**	**platform1**	**platform2**	**connection**
1	None	None	On	15000	**data**									
2	None	configSet	On	65000	**data**						On			On
3	None	event	On	0	event			On			On			On
4	On	None	On	None	**data**						On			On
5	On	configSet	On	15000	**event**						On			On
6	On	event	On	65000	event			On			On			On
7	None	configSet	On	None	**event**						On			On
8	On	None	On	0	**event**						On			On
9	None	None	On	65000	**event**									
10	None	configSet	On	0	**data**						On			On
11	None	event	On	None	event			On			On			On
12	None	event	On	15000	event			On			On			On

Table 18 - Configuration Element Table after processing the information element.

The next element (step 8) to be processed is the 'data' element. As this element is one of the choices in the 'information' choice container it can only be turned on when the 'information' element has chosen it. As the 'information' element and the 'data' element have a direct dependency the pairs between 'information' and 'data' are not considered by pairwise. For example, if 'data' is set to On then 'information' must be given the value of data. Alternatively, if 'information' has a value of data then 'data' must be On. All valid pairs that need to be covered for the 'data' element are provided in the table, table 19 (step 9).

setRef	data	configRef	data	configSet	data	id	data
None	None	None	None	On	None	None	None
None	On	None	On	On	On	None	On
On	None	configSet	None			0	None
On	On	configSet	On			0	On
						65000	None
						65000	On
						15000	None
						15000	On

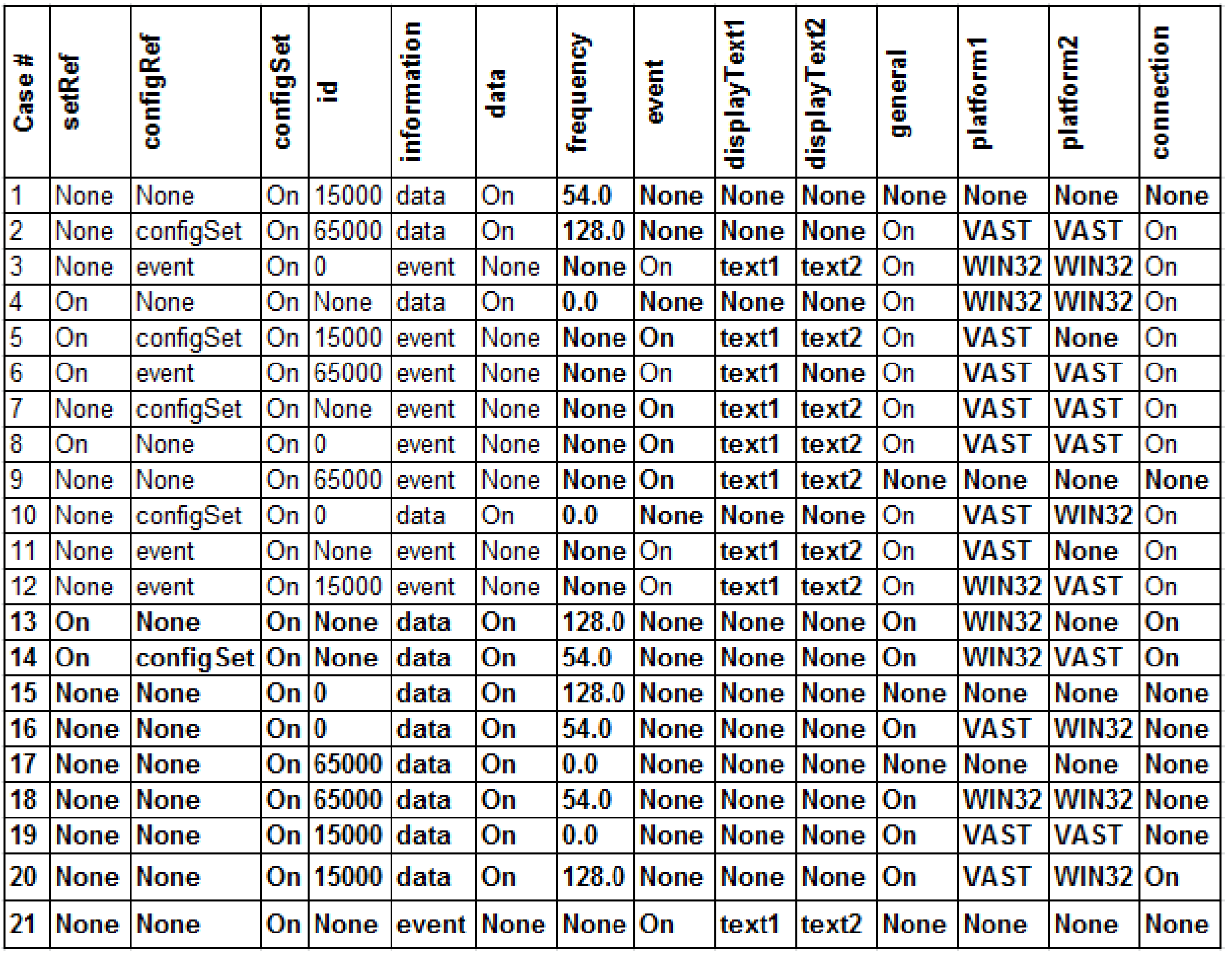
Table 19 - All valid pairs for the data element.

In table 19 there are no pairings with 'configRef' set to event. This is because it is not possible to have both 'event' and 'data' in the same configuration as they are both choices within the same choice container. The pairing (configRef: event, data: None) is implicitly covered in that while referencing 'event', 'data' must have a value of None. None of the pairs in the table have been covered so far (step 10) so all are processed by the pairwise algorithm. As 'data' is a choice within 'information' it can only be turned on if 'information' has chosen it. This is what step 12 caters for. Step 12 also handles the case where the parent is None so the value must be None or when trying to add a value for a reference that references an incorrect choice in the row in which case it skips this value and adds another value later on. Setting values for 'data' to satisfy this requirement covers all the pairs in table 19 resulting in the Configuration Element from table 20.

Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000	data	**On**								
2	None	configSet	On	65000	data	**On**					On			On
3	None	event	On	0	event	**None**		On			On			On
4	On	None	On	None	data	**On**					On			On
5	On	configSet	On	15000	event	**None**					On			On
6	On	event	On	65000	event	**None**		On			On			On
7	None	configSet	On	None	event	**None**					On			On
8	On	None	On	0	event	**None**					On			On
9	None	None	On	65000	event	**None**								
10	None	configSet	On	0	data	**On**					On			On
11	None	event	On	None	event	**None**		On			On			On
12	None	event	On	15000	event	**None**		On			On			On

Table 20 - Configuration Element Table after processing the data element.

The pairwise algorithm carries out the same processing for the rest of the elements from the Definition Element Table. Table 21 is the complete Configuration Element Table required to cover all pairs for the Dsm example. In the ‘frequency’ column it can be seen that some of the rows have a value of None. None is not one of the possible values for ‘frequency’ according to the Definition Element Table. However, as ‘frequency’ is a child of ‘data’, in the cases where ‘data’ has been given a value of None then ‘frequency’ must also be None. This corresponds to restriction number 1 in the list of restrictions relating to the values that are invalid within a row.



Case #	setRef	configRef	configSet	id	information	data	frequency	event	displayText1	displayText2	general	platform1	platform2	connection
1	None	None	On	15000	data	On	54.0	None	None	None	None	None	None	None
2	None	configSet	On	65000	data	On	128.0	None	None	None	On	VAST	VAST	On
3	None	event	On	0	event	None	None	On	text1	text2	On	WIN32	WIN32	On
4	On	None	On	None	data	On	0.0	None	None	None	On	WIN32	WIN32	On
5	On	configSet	On	15000	event	None	None	On	text1	text2	On	VAST	None	On
6	On	event	On	65000	event	None	None	On	text1	None	On	VAST	VAST	On
7	None	configSet	On	None	event	None	None	On	text1	text2	On	VAST	VAST	On
8	On	None	On	0	event	None	None	On	text1	text2	On	VAST	VAST	On
9	None	None	On	65000	event	None	None	On	text1	text2	None	None	None	None
10	None	configSet	On	0	data	On	0.0	None	None	None	On	VAST	WIN32	On
11	None	event	On	None	event	None	None	On	text1	text2	On	VAST	None	On
12	None	event	On	15000	event	None	None	On	text1	text2	On	WIN32	VAST	On
13	On	None	On	None	data	On	128.0	None	None	None	On	WIN32	None	On
14	On	configSet	On	None	data	On	54.0	None	None	None	On	WIN32	VAST	On
15	None	None	On	0	data	On	128.0	None	None	None	None	None	None	None
16	None	None	On	0	data	On	54.0	None	None	None	On	VAST	WIN32	None
17	None	None	On	65000	data	On	0.0	None	None	None	None	None	None	None
18	None	None	On	65000	data	On	54.0	None	None	None	On	WIN32	WIN32	None
19	None	None	On	15000	data	On	0.0	None	None	None	On	VAST	VAST	None
20	None	None	On	15000	data	On	128.0	None	None	None	On	VAST	WIN32	On
21	None	None	On	None	event	None	None	On	text1	text2	None	None	None	None

Table 21 - Final Configuration Element Table for the Dsm example.

In order to create the ARXML files that are used by the SCG, the above table needs to be converted from a Configuration Element Table into ARXML. First, containers for external references need to be added. ‘setRef’ is an external reference so for the rows where this has a value of On additional columns are added containing the referenced container from the Drm module. This updated Configuration Element Table is then processed by the Configuration Creator as described section 4, which is responsible for the conversion from the Configuration Element Table into ARXML. This results in 21 AUTOSAR configurations which cover all pair combinations between all element values from the Definition Element Table.

6. Experiments

This section describes the experiments that were conducted in order to compare the two generators. It presents the results of these experiments and the conclusions we draw from them.

Experiment 1: Number of crashes based on set number of configurations

In one run the pairwise generator creates all the configurations needed in order to cover all pairs of elements values. It is expected that the pairwise technique, according to McGregor (McGregor, 2010), can find at least 90% of the faults that an exhaustive method can. The random generator is similar to an exhaustive technique except that it can create duplicates. It creates as many configurations as it is asked to, or as many configurations as it can create in a given time. While the pairwise generator never creates duplicate configurations, the random generator may create duplicates. The pairwise generator, taking a parameter definition file as input, generates exactly the same number of configurations for every run. To compare the efficiency of the two algorithms, the random algorithm is asked to create the same number of configurations that pairwise does in one run. Both sets of configurations are run through the SCG and the number of crashes are counted for each generator and finally compared.

Experiment 2: Number of crashes based on set amount of time

This second experiment is similar to the first experiment with the exception that in this case the random generator is not limited to a fixed number of configurations, but to a set amount of time, e.g. four hours. The number of configurations for a module(s) generated with the random algorithm is much larger than the number of pairwise configurations for the same module(s). The pairwise generator is expected to create all the configurations and run them through the SCG before the time elapses (giving it a reasonable time). For the pairwise generator there is no need to run again, even if the time permits it, because it will never produce anything different. The random generator will try to create as many configurations as possible and run each through the SCG in that set amount of time. The number of crashes can be compared when running the pairwise and the random configurations through the SCG.

Experiment 3: Overhead of time taken to generate configurations

The random generator is expected to be faster in generating the same number of configurations needed to complete the pairwise algorithm. Running these configurations through the SCG is expected to take much more time than creating them. With this experiment we want to see if the time difference between the two generators is noteworthy even after considering the time needed to run the configurations of each generator through the SCG.

Setup of the experiments

The experiments were conducted for three of the Basic Software modules with different sizes and complexity levels. The complexity is higher for modules that contain elements such as choice-containers and/or references. It increases especially for the pairwise algorithm that needs to cover all pair combinations of elements values. For a choice container, all pairs that include one of the choices that is not selected in a given row cannot be satisfied within the same row. For references, the complexity increases because of all the elements that need to be added when setting a value to a reference, e.g. its parents, its referenced element. Most of the pairs for these elements that need to be added are not satisfied in the current configuration so the algorithm needs to create many configurations in order to satisfy all the pairs. Mainly, the complexity increases because of the invalid pairs/values described in section 5.3.1.

The first module is called Development Error Tracer (Det) and it is a small size module with a low complexity because, as mentioned above, it does not contain any reference or choice containers. This module includes 7 elements (2 containers and 5 parameters). The parameter definition in ARXML format for the Det module is available in Appendix A. The second module is called Function Inhibition Manager (FiM) and it is a medium size module that contains 35 elements (8 containers and sub-containers, 1 choice container with 2 choices, 15 parameters and 9 references). More details about this module can be found on the AUTOSAR website (AUTOSAR FiM, 2011). The last module is called Diagnostic Event Manager (Dem) and it is a large size module with 195 elements (34 containers and sub-containers, 2 choice-containers with 3 choices each, 120 parameters and 33 references). Some of these elements have upper multiplicities set to 255, 65535 or even infinite and value ranges up to 32768, 65536 or even around 16 million. Detailed information about the Dem module can be found on the AUTOSAR website (AUTOSAR Dem, 2011).

For all of the modules used in these experiments we have set a limit value equal to 5 for the elements that can have more instances. For example, if a container had a lower multiplicity of 0 and an upper multiplicity of infinite, we have changed the value of the upper multiplicity to 5. If a parameter had a lower multiplicity of 10 and an upper multiplicity of 255, we have set the upper multiplicity to 15. If the difference between the lower and the upper multiplicity of an element was less than 5 we left them unchanged. To gain understanding about the complexity of the Dem module, the pairwise generator needs to process 1288 elements when creating the Dem configurations with a limit value set to 5. For the example presented in section 5.2.3. the pairwise generator had to process only 14 elements. This limit value was chosen in cooperation with Mecel AB.

As previously described, the SCGs have two components of interest for the purpose of testing. The first is a component that checks whether the provided configuration conforms to the rules in the modules specification. The second component generates the source code. In order to pass the first component of the SCGs we had to constrain our configurations.

For the Det module, its structure has been modified to satisfy the constraints. The new structure, as described in figure 37, was limited using a limit file. The difference is that the Dlt parameter has a lower multiplicity of 1 whereas the original Dlt had a lower multiplicity of 0. We made this change due to a constraint that expects a Dlt parameter to be present.

For FiM we implemented a constraints handler that makes sure the configurations created by both pairwise and random are satisfying the constraints defined for the FiM module. An example of a constraint defined for the FiM module says that the value of the FiMFunctionId integer parameter belonging to the FiMFID sub-container shall start from 1 and be consecutive throughout the entire configuration. So if we create 3 instances of the FiMFID sub-container in a configuration, the first sub-container shall have its FiMFunctionId set to 1, the second container shall have it set to 2 and the last one to 3.

For the Dem module a constraint handler was not implemented because the Dem SCG was checking for over two hundred constraints to be satisfied by the configurations. While we have provided a constraints framework, the scale of constraining Dem is too great to accompany this thesis work and can make up part of a future thesis project. This means that in our experiments the Dem SCG was expected to signal that the given configurations are valid, but not working configurations. It was also possible that the Dem SCG could crash when running the constraints checker part of the SCG.

The structure of the new parameter definition file for the Det module is described in figure 37. As mentioned earlier in this section, the limitations have been applied to this parameter definition so that the lower multiplicity of the Dlt parameter was set to 1 and the upper multiplicity of the ErrorHook parameter was set to 5. The configuration cases created by pairwise for Det are used in a number of experiments so the list of cases is provided in table 22. For FiM and Dem the number of pairwise configurations was too large for us to be able to display it in this paper.

The pairwise generator always produces the same result for a given parameter definition with small differences regarding the values of the elements. For an integer or a float parameter, the pairwise generator will assign three values besides None. The first value is the minimum value that the parameter can have according to the parameter definition. The second value is the maximum value that it can have according to the same parameter definition. The third value is obtained by randomly choosing a value between the minimum and the maximum values. For a string parameter that has defined a regular expression, the pairwise generator assigns one random string value that conforms to the regular expression. This is done by using a java library called Xeger (Xeger, 2009). Giving the string all possible values is unnecessary from a testing perspective and practically infeasible. Even if the results of the pairwise algorithm differ in these manners, they are still created using the same algorithm every time and they always contain the same number of configurations with the elements values chosen in the same way. These randomly chosen values do not influence the results, according to Mecel AB, and we can say that pairwise produces the same configurations in each run.

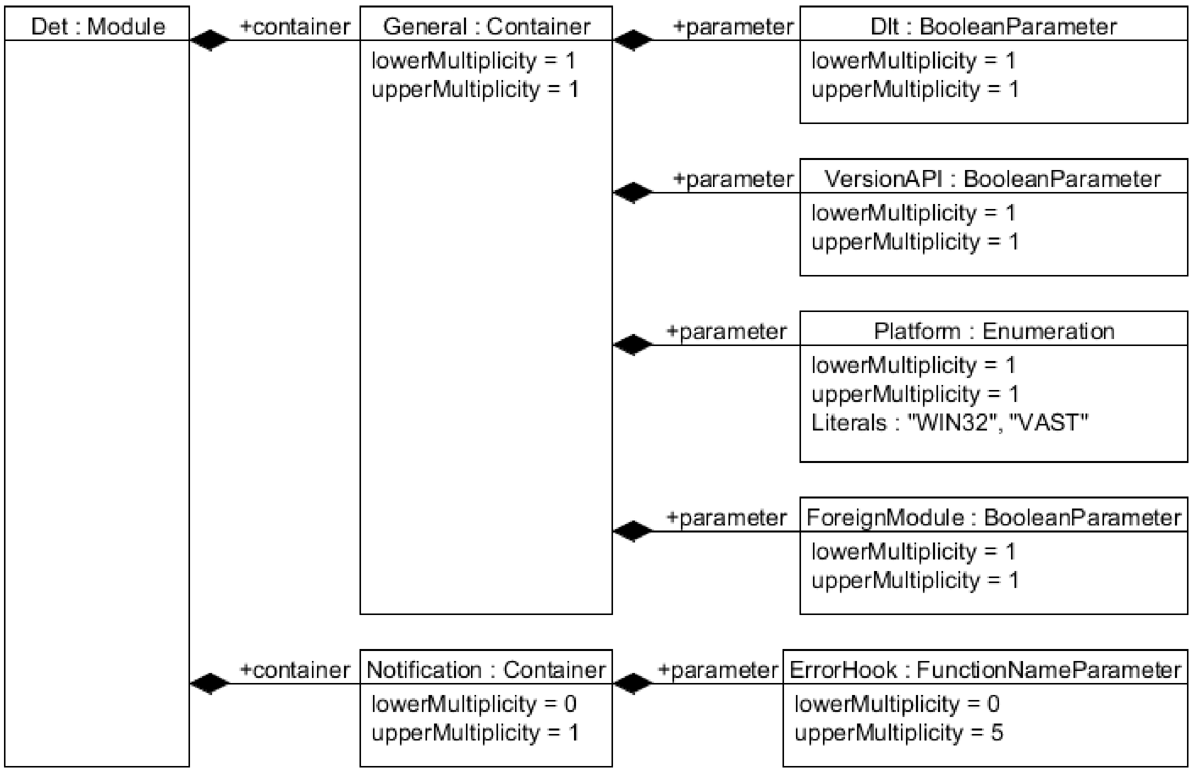


Figure 37 - The parameter definition structure of the Det module.

Case #	General	Dlt	VersionAPI	Platform	ForeignModule	Notification	ErrorHook	ErrorHook_1	ErrorHook_2	ErrorHook_3	ErrorHook_4
1	On	1	0	VAST	0	On	ErrorHook_0	ErrorHook_1	ErrorHook_2	ErrorHook_3	ErrorHook_4
2	On	0	1	WIN32	1	None	None	None	None	None	None
3	On	1	1	VAST	1	On	None	ErrorHook_1	None	ErrorHook_3	None
4	On	0	0	VAST	0	None	None	None	None	None	None
5	On	1	0	WIN32	0	On	ErrorHook_0	None	ErrorHook_2	None	ErrorHook_4
6	On	1	1	WIN32	0	None	None	None	None	None	None
7	On	1	0	WIN32	1	On	ErrorHook_0	ErrorHook_1	ErrorHook_2	ErrorHook_3	None
8	On	0	1	WIN32	1	On	ErrorHook_0	ErrorHook_1	ErrorHook_2	ErrorHook_3	ErrorHook_4
9	On	1	1	WIN32	1	On	None	ErrorHook_1	ErrorHook_2	None	ErrorHook_4
10	On	1	1	WIN32	1	On	ErrorHook_0	ErrorHook_1	None	ErrorHook_3	ErrorHook_4
11	On	1	1	WIN32	1	On	ErrorHook_0	None	ErrorHook_2	ErrorHook_3	ErrorHook_4

Table 22 - Configuration Element Table generated by pairwise for Det.

The experiments described in this section were run on a computer with the following characteristics:

- Processor: Intel(R) Core(TM)2 Duo CPU T9400 @2.53 GHz 2.54 GHz
- Installed memory (RAM) 4.00 GB
- System type: Windows 7 64-bit Operating System

The version of Java was 1.7.0_10 and the VM arguments for running our application were set to: -Xms128m -Xmx3000m -Xss4m -XX:MaxPermSize=256m.

6.1. Experiment 1: Number of crashes based on set number of configurations

The first experiment was conducted in order to compare the number of crashes based on set number of configurations, as described in the beginning of this chapter.

6.1.1. Design

The following steps have been followed when designing the first experiment for each of the three modules Det, FiM and Dem.

1. Run the pairwise generator to create all configuration cases. For the Det module, this gives the cases in table 22. Store the number of configurations.
2. Run all pairwise configurations through the SCG.
3. Analyse the output of the SCG to identify crashes from pairwise configurations. Store the number of crashes and the name of the configuration that crashed. Also identify and store the number of handled errors.
4. Run the random generator to create the same number of configurations as pairwise, i.e. 11 for Det.
5. Run all random configurations through the SCG.
6. Analyse the output of the SCG to identify crashes from random configurations. Store the number of crashes and the name of the configuration that crashed. Also identify and store the number of handled errors.
7. Compare the number of crashes identified by the pairwise generator with the number of crashes identified by the random generator.

Notes:

- To identify crashes the output of the SCG is automatically read and analysed based on some expected printing messages according to the implementation of the SCG. If none of the expected messages that signal a handled error or a successful run are displayed, then we categorize this run of the SCG as a crash. A message containing an unhandled error is also counted as a crash.

6.1.2. Results

Table 23 shows the averaged results of running the first experiment 10 times. The experiment is run for each module 10 times and each row represents one module.

Nr.	Module	Configurations	Crashes from Random	Crashes from Pairwise	Errors from Random	Errors from Pairwise
1	Det	11	0	0	0	0
2	FiM	504	0	0	42	35
3	Dem	2431	0	0	2431	2431

Table 23 - Averaged Crash Data for each module in experiment 1.

6.1.3. Discussion

This section discusses each row from table 23 and gives a conclusion related to the first experiment.

For the first row, the module under experiment was Det. The pairwise generator produced 11 configurations. As a result 11 random configurations were also generated. When the pairwise configurations were run through the SCG no crashes and no errors were identified. When the random configurations were run through the SCG no crashes and no errors were identified either.

For the second row, the module under experiment was FiM. The pairwise generator produced 504 configurations. As a result 504 random configurations were also generated. When the pairwise configurations were run through the SCG no crashes and 35 errors were identified. When the random configurations were run through the SCG no crashes and an average of 42 errors were identified.

For the third and final row, the module under experiment was Dem. The pairwise generator produced 2431 configurations. As a result 2431 random configurations were also generated. When the pairwise configurations were run through the SCG no crashes and 2431 errors were identified. When the random configurations were run through the SCG no crashes and an average of 2431 errors were identified.

As the Det module is less complex in comparison to other modules, we did not expect to identify crashes or errors in the Det SCG at this stage.

For the FiM module, the results indicate that the FiM SCG was tested using working configurations which test both the constraints checking and the source code generation aspects of the SCG. No crashes were reported after applying both the pairwise and the random techniques showing that both generators are comparable and that the FiM SCG is robust at crash handling. When analysing the numbers of errors found for the FiM module and their print messages, we saw that the pairwise generator discovered the same error in 35 of the 504 configurations. The random generator also discovered the same error that pairwise found, but in an average of 42 configurations out of 504. This error that both generators encountered was handled by the FiM SCG. It had to do with one of the constraints that we have applied when creating configurations for the FiM module. This means that the FiM SCG was not validating the configuration against this constraint in the specified way. Regarding this, Mecel AB confirmed that the FiM SCG contained a bug and have raised a report to have it fixed.

As for the Dem module, both the pairwise and the random configurations were valid, but not working configurations since they were not constrained, so the SCG found handled errors in all of them and it did not crash. This result shows that the Dem SCG is robust at error and crash handling, since there were no unhandled errors. It also shows that the two generators always created valid configurations and that they did not crash even when creating a large number of configurations for a high complexity module such as Dem.

6.2. Experiment 2: Number of crashes based on set amount of time

The second experiment was conducted in order to compare the number of crashes based on set amount of time, as described in the beginning of this chapter.

6.2.1 Design

The following steps have been followed when designing the second experiment for each of the three modules Det, FiM and Dem.

1. Set the amount of time the random generator should run for. This time is only needed by the random generator in order to know when to stop creating configurations and running them through the SCG. The pairwise generator does not stop until it creates all configurations it needs to create. If we would stop the pairwise generator before it finishes, no results will be obtained. There is also no need to run the pairwise generator multiple times for the same module in a given time, because, as mentioned earlier, it always creates the same result.

2. Run the pairwise generator and run the created configurations through the SCG.

3. Analyse the SCG output and count the number of crashes from the pairwise configurations. The names of the configurations that crash are also provided. Also identify and store the number of handled errors.

4. Run the random generator and run the created configurations through the SCG for the amount of time specified in step 1. This amount of time has been increased as the complexity of the modules increased. For Det it was set to one hour, for FiM to two hours and for Dem to four hours. The time was increased in order to give the random generator at least as much time as the pairwise generator needs to run once. The time needed to run a set of experiments during a nightly build was also considered in this equation.

5. Analyse the SCG output and count the number of crashes from the random configurations. The names of the configurations that crash are also provided. Also identify and store the number of handled errors.

6. Compare the number of crashes identified by the pairwise generator with the number of crashes identified by the random generator.

6.2.2. Results

The table below shows the averaged results of running the experiment 10 times. The experiment is run for each module 10 times and each row represents one module.

Nr.	Module	Pairwise Config.	Random Config.	Crashes from Pairwise	Crashes from Random	Errors from Pairwise	Errors from Random	Time
1	Det	11	2813	0	0	0	0	1h
2	FiM	504	5189	0	0	35	512	2h
3	Dem	2431	10686	0	0	2431	10686	4h

Table 24 - Averaged Crash Data for each module in experiment 2.

6.2.3 Discussion

This section discusses each row from table 24 and gives a conclusion related to the second experiment.

For the first row, the module under experiment was Det and the given amount of time was one hour because of the small size of this module. The pairwise generator produced 11 configurations, ran them through the SCG and stopped before the time elapsed. The random generator produced an average of 2813 configurations and ran them through the SCG in one hour. When the pairwise configurations were run through the SCG no crashes and no errors were identified. When the random configurations were run through the SCG no crashes and no errors were identified either.

For the second row, the module under experiment was FiM and the given amount of time was two hours because of the medium size of this module. The pairwise generator produced 504 configurations, ran them through the SCG and stopped before the time elapsed. The random generator produced an average of 5189 configurations and ran them through the SCG in two hours. When the pairwise configurations were run through the SCG no crashes and 35 errors were identified. When the random configurations were run through the SCG no crashes and an average of 512 errors were identified.

For the third row, the module under experiment was Dem and the given amount of time was four hours because of the large size of this module. The pairwise generator produced 2431 configurations, ran them through the SCG and stopped before the time elapsed. The random generator produced an average of 10686 configurations and ran them through the SCG in four hours. When the pairwise configurations were run through the SCG no crashes and 2431 errors were identified. When the random configurations were run through the SCG no crashes and 10686 errors were identified.

For the pairwise generator, we did not expect any crashes to be identified when running the SCGs since the same configurations were generated in the first experiment and they did not crash.

In this experiment the random generator was not limited to create the same number of configurations as the pairwise generator. This gave the random generator a chance to create as many configurations as it could in a given time so that it increases its chances of identifying crashes in the SCGs. This was not the case for the second experiment as no crashes were identified this time either, so the random generator did not outperform the pairwise generator in this experiment. This helps to strengthen the conclusion given in the first experiment with regards to the robustness of the SCGs under test at crash handling.

Because of the low complexity of the Det module, we did not expect any crashes or errors at this stage also. For the medium size module, called FiM, both generators found the same error as in the first experiment. The pairwise generator found it in 35 of the 504 configurations and the random generator found it in an average of 512 out of 5189 configurations. For the large size module, called Dem, the SCG did not fail this time either at handling all the errors in each configuration, which shows again the robustness of the Dem SCG at error handling.

6.3. Experiment 3: Overhead of time taken to generate configurations

The third experiment was conducted in order to determine if an overhead is introduced by using the pairwise algorithm, as described in the beginning of this chapter.

6.3.1 Design

The following steps have been followed when designing the third experiment for each of the three modules Det, FiM and Dem.

1. Run the pairwise generator to create all configuration cases. For Det, this gives the cases in table 22. Store the time taken to create the pairwise configurations.

2. Run all pairwise configurations through the SCG. Add the time taken to run all configurations through the SCG to the time taken to create the configurations. Store the result.

3. Run the random generator to create the same number of configurations as pairwise. Store the time taken to create the random configurations.

4. Run all random configurations through the SCG. Add the time taken to run all configurations through the SCG to the time taken to create the configurations. Store the result.

5. Divide the times taken to create the pairwise configurations with the time taken to create the random configurations. This gives the factor difference between the two values.

6. Divide the times taken to create the pairwise configurations and run them through the SCG with the time taken to create the random configurations and run them through the SCG. This gives the factor difference between the two values.

6.3.2 Results

The table below shows the averaged results of running the experiment 10 times. The experiment is run for each module 10 times and each row represents one module.

Nr.	**Module**	**A:Pairwise conf. time (s)**	**B:Random conf. time (s)**	**C:Pairwise conf. & SCG time (s)**	**D:Random conf. & SCG time (s)**	**A/B**	**C/D**
1	Det	0.31	0.07	13.03	12.57	4.43	1.04
2	FiM	37.07	5.6	1034.82	997.21	6.62	1.04
3	Dem	10224.25	68.17	13347.36	3175.45	149.98	4.2

Table 25 - Averaged generation and SCG times for each module.

6.3.3. Discussion

This section discusses each row from table 25 and gives a conclusion related to the third and final experiment.

For the first row, the module under experiment was Det. The pairwise generator created 11 configurations in an average time of 0.31 seconds. The random generator created the same number of configurations, i.e. 11, in an average time of 0.07 seconds. Dividing these two values, we can see that the random generator was in average 4.43 times faster than the pairwise generator in these 10 runs of the experiment. The pairwise generator created 11 configurations and ran all of them through the SCG in an average time of 13.03 seconds. The random generator created the same number of configurations, i.e. 11, and ran all of them through the SCG in an average time of 12.57 seconds. Dividing these two values, the factor difference is 1.04. This means that the random generator together with the SCG was still faster than the pairwise generator together with the SCG for the Det module in these 10 runs of the experiment. However, as the factor difference is so close to 1, the overhead introduced by pairwise is not noteworthy. The times recorded in this experiment are included in Appendix B.

For the second row, the module under experiment was FiM. The pairwise generator created 504 configurations in an average time of 37.07 seconds. The random generator created the same number of configurations, i.e. 504, in an average time of 5.6 seconds. Dividing these two values, we can see that the random generator was in average 6.62 times faster than the pairwise generator in these 10 runs of the experiment. This factor difference is larger than the 4.43 factor difference for the Det module, because of the complexity added by the choice container and references of the FiM module, as explained in the introduction of this chapter. The pairwise generator created 504 configurations and ran all of them through the SCG in an average time of 1034.82 seconds, i.e. approximately 17 minutes. The random generator created the same number of configurations, i.e. 504, and ran all of them through the SCG in an average time of 997.21 seconds, i.e. approximately 16 and a half minutes. Dividing these two values, the factor difference is 1.04. This means that the random generator together with the SCG was still faster than the pairwise generator together with the SCG for the FiM module in these 10 runs of the experiment. However, as the factor difference is so close to 1, the overhead introduced by pairwise is not noteworthy.

For the third and final row, the module under experiment was Dem. The pairwise generator created 2431 configurations in an average time of 10224.25 seconds, i.e. almost 3 hours. The random generator created the same number of configurations, i.e. 2431, in an average time of 68.17 seconds. Dividing these two values, we can see that the random generator was 149.98 times faster than the pairwise generator in these 10 runs of the experiment. The pairwise generator created 2431 configurations and ran all of them through the SCG in an average time of 13347.36 seconds, i.e. almost 4 hours. The random generator created the same number of configurations, i.e. 2431, and ran all of them through the SCG in an average time of 3175.45 seconds, i.e. almost one hour. Dividing these two values, the factor difference is 4.2. This means that the random generator together with the SCG was still faster than the pairwise generator together with the SCG for the Dem module in these 10 runs of the experiment. It is worth noting here that the Dem module was unconstrained, so when running the SCG, all the configurations were rejected during the constraints checker component of the SCG. As the Dem SCG did not run to completion, the times recorded do not reflect the total time needed to test the SCG. This means that the factor difference of 4.2 is worse than if the SCG ran to completion.

When comparing the times needed to create the configurations for each module in these 10 runs of the experiment we notice that there was an overhead impact introduced by running the pairwise generator. However, when the time taken to run the configurations through the SCG was included, the overhead introduced by pairwise decreased.

6.4. Conclusion

The following table, table 26, shows the types of configurations that each generator succeeded in creating for each of the three modules. The generators produced valid and working configurations for the Det and FiM modules and valid but not working configurations for the Dem module.

Type	Det	FiM	Dem
Valid	Yes	Yes	Yes
Working (Constraints Applied)	Yes	Yes	No

Table 26 - Configuration Types for modules.

Something else to consider when comparing the two generators is the number of parameter definition elements that each generator can cover. The pairwise generator builds all possible pairings. Due to this, no element will be left uncovered. For example in the table 22 the Notification container can have the values None and On, both of which are represented in the configurations. The random generator takes the lower and upper multiplicity of an element and it randomly generates a number (inclusive) between them. This represents the number of times this element is added to the configuration. In the example using the Notification container the lower multiplicity is 0 and the upper multiplicity is 1. It is possible that the random generator will always randomly pick a lower multiplicity of 0. As a result the Notification container could never be represented in any configuration. We have counted the number of covered elements by both generators during the first experiment and we have found that they both had 100% coverage, so the random generator did not skip any elements for the modules under test. The averaged results are illustrated in table 27.

Nr.	Module	Nr. of elements	Elements covered by Random	Elements covered by Pairwise
1	Det	7	7	7
2	FiM	35	35	35
3	Dem	195	195	195

Table 27 - Averaged number of elements covered by each generator.

When comparing the number of crashes that each generator can identify, our first two experiments did not find a difference since none of the generators uncovered crashes in the SCGs. This showed that all the SCGs under test were robust at crash handling.

When comparing the number of errors handled by each SCG, we found that both generators uncovered a problem in the FiM SCG. The random generator found this error in more configurations than the pairwise generator did, but this does not prove that random is better from this point of view, since we talk about a single error and both generators found it. We also looked at which generator found the error faster and we found that pairwise, as it always creates the same result, found the error in the second configuration every time, but it had to create all the configurations first before it can run any of them through the SCG. The random generator found the error in the fifth configuration or even later, but it found it faster than the pairwise generator because the random generator ran each configuration through the SCG right after it was created. This result does not prove that random is faster from this point of view, since both generators ran together on a nightly builder and the error was recorded after both of them finished running.

Using the pairwise generation technique introduced an overhead when considering the time taken to create the same number of configurations as the random generator in the 10 runs of the experiment. This overhead decreased for all modules under test when considering the time to run all the configurations through the SCGs. This happened mainly because the time taken to run the configurations through the SCGs is greater than the time needed to create the configurations. For the Det and the FiM module, the overhead introduced by using the pairwise algorithm decreased to a level that is not noteworthy. For the Dem module, it took the pairwise generator more time to create all the configurations than it took to run them through the SCG. This happened because the Dem SCG never needed to generate any code, since all inputs were rejected because of unsatisfied constraints.

Another important aspect that we have to take into consideration when comparing the two generators is the fact that the pairwise generator only needs to run once to create all the configurations that cover all pairs of element values.

One of the limitations of the experiments we conducted in this study is the number of modules under test. One may argue that only three Basic Software modules are not enough to illustrate the real difference between the two generation techniques. Since it takes one nightly build to run a set of experiments for each of the three modules, we needed 10 nightly builds to gather this data. We agree that more tests could uncover more benefits when using one of the two generators. We chose our modules trying to cover different size and complexity levels. That is why we chose a small and low complexity module such as Det, a medium size and medium complexity module as FiM and a large and high complexity module as Dem.

Another limitation of the experiments has to do with the number of times we ran each experiment. The pairwise generator is deterministic so given more runs it would not produce different results to the ones we recorded. However the random generator may produce different results if run more times. We tried to mitigate this by giving the random generator more time to run with the second experiment.

In line with the limitations for these experiments is also the fact that the configurations for the Dem module were not constrained and the time taken to run the Dem configurations through the SCG is actually less than it should be, as only the constraints checker component of the SCG ran and never the code generation component. This was useful for the first and second experiment, since the configurations tested the robustness to error handling of the Dem SCG.

7. Related Work

This chapter discusses related work in the area of combinatorial testing, software product line testing and integration testing. It is divided into sections which serve to group the related work into categories. The first category is software product lines, an area of interest due to the similarities with AUTOSAR parameter definitions. The second category is pairwise algorithms with literature for algorithms related to our pairwise algorithm and alternative algorithms. The third category contains literature on additional methods that could be used to improve random generation.

7.1. Software Product Lines

This section describes the related work from software product lines literature and how this literature relates to this thesis.

McGregor's paper (McGregor, 2010) discusses software product lines and how they can be used for better quality, productivity and cycle times. Changes need to be made in the approach to testing to help achieve these improvements. There is much variability across a software product line so McGregor discusses how to handle this variability when testing and different testing techniques that can be applied at different stages of the development process. McGregor identifies the main challenges present when testing a software product line. These are:

- *Variability*: The more variability that is present within the product line the more resources will be required for testing.

- *Emergent Behaviour*: When components are combined some unexpected behaviour may occur that is not present in the individual components. It is difficult to have a reusable test case for this.

- *Creation of Reusable Assets*: Test cases and test data can be broken into smaller chunks in order to facilitate their reuse in more places. This reuse comes at the cost of more planning time in defining the smaller chunks and more management needed for the increased number of artefacts.

- *Management of Reusable Assets*: Certain information needs to be managed relating to reusable assets. This includes traceability between all parts of an asset, where the assets should be stored and when an asset can be used. Configuration management systems can be used to achieve this.

An organization uses software product lines to identify the common features across all of their products to allow them to reuse these common features. Tests are created in parallel with the features making them reusable for common features and traceable overall. The level of testing required depends on the domain. Medical software needs to be tested more rigorously than video game software. The method for producing the tests should follow the same method for producing the production code. In developing the tests in line with the product the tests are inherently reusable within that context.

The variability that exists within software product lines introduces a number of implications on testing:

- *Variation occurs at specified points:* Testing needs to be able to respond to these variation points, selecting the correct tests.

- *Product variation means there will be test variation:* Tests usually have the same variation points as the product and tests should be related to the product they test, e.g. by using a builder that builds the product and tests together.
- *The product and test goals should match:* The qualities expected from the product should also be expected from the tests.
- *Variants are bound at a point in time:* The tests should also bind at the correct time.
- *All bindings in the product must also be present in the tests:* Test cases may need clever techniques to achieve this e.g. for dynamically bound variants.
- *It is important to manage the amount of variation:* Variation has results in more combinations to be considered in testing. Each variant should be considered carefully before being added.
- *Common components:* Commonality allows for less retesting and for reusing tests.

McGregor discusses combinatorial testing in more detail. Due to the variability of software product lines exhaustive testing of all combinations could require billions to trillions of test cases. Using combinatorial techniques a high level of coverage (up to 90%) can be achieved with 30 - 60 test cases. The pairwise technique can be used to cover pairs of parameters, 3-way, 4-way up to n-way. Pairwise testing can be used for creating a configuration for a product.

The configurations we create are based on a parameter definition. This definition can contain mandatory and optional elements making it similar to a product line. Due to the stark similarities between our requirements and McGregor's statements we decided to investigate the pairwise technique further.

McGregor also recommend identifying areas in which variability can be reduced before applying the pairwise technique. We have employed this concept in the following ways:

- Use a limit file and limit value to put limitations on a parameter definition.
- Limit value ranges to the max value, min value and one value in between.

The paper from Ye et al. (Ye et al., 2005) is based in the area of software product lines. The authors propose a feature-oriented method for modelling feature dependencies and variability in a software product line. Variability gives the opportunity to customize software product lines. The features of a software product line are never stand-alone and they also depend on or influence other features.

The authors' approach supports detection of conflicting features and consists of a feature tree view and a feature dependency view. The latter represents the main interest of the authors for their study. It consists of a set of individual dependency diagrams and a matrix that comprises all these dependencies. An individual dependency diagram describes all the direct and indirect dependencies of one feature.

The authors define three hierarchical and three non-hierarchical relationships between features. The hierarchical relationships are *composition*, *generalisation/specialisation* and *variation point*. Composition implies that a certain feature consists of another feature. A generalisation/specialisation relationship between two features means that one feature is a generalised feature of the other feature. The variation point relationship implies that one feature is optional and it is a child of another feature. The non-hierarchical relationships defined by the authors are *requires*, *excludes* and *impacts*. The *requires* relationship between two features means that one feature requires the other feature. The *excludes* relationship is bi-directional and it implies that two features exclude each other. The *impacts* relationship means that one feature impacts on another feature.

For the individual dependency diagrams, the authors use a set of UML-based notations and a custom notation for the variation point. The other component of the feature dependency view, the feature dependency matrix, is developed to store the dependencies between all features of a product line. It is a n x n matrix, where n is the number of features, and it contains in each cell M(i,j) the dependency between the feature contained in row i with the feature contained in column j. The authors also encode these dependencies with numerical values, e.g. 1 for requires, 100 for excludes, and include an algorithm for generating the individual dependency views, which they decompose into a set of forward-to and backward-to dependency diagrams. In the tree view, they distinguish between a mandatory and a variable feature, by assigning the <<variant>> stereotype to the variable features.

In the future work of our study, the feature dependency matrix could be a good way to store and manipulate the information about dependencies between different Configuration Elements and rules in the Basic Software modules.

7.2. Pairwise Algorithms

This section contains the literature related to pairwise algorithms. At a high level there are two categories of pairwise algorithm, those using an all pairs approach and those using orthogonal and covering arrays. The literature for the all pairs approach is directly related to our thesis. The literature on orthogonal and covering arrays is presented as an alternative considered.

All-Pairs

The Bellcore paper (Cohen et al., 1996) from Cohen et al. describes a combinatorial design method used for testing. During some experiments, the method presented good code coverage and the ability to uncover faults that were not detected by a standard process. They use this method for selecting fewer combinations of parameters to test, since all the possible combinations are often too many. Using a system called AETG, a tester can specify what are the parameters that need to be tested and their possible values. They also mention that a tester can specify constraints between the different test parameters. According to the authors the most used combinatorial methods are pairwise and triple (3-wise). An empirical study done at Bellcore revealed that the most faults were caused by an incorrect single value or by an interaction of two values, i.e. a pair. A code coverage study also shows that pairwise is sufficient for obtaining good code coverage. Multiple comparisons were done between the number of test sets needed for all possible combinations of parameters and the number of test sets covering the pairwise combinations. In one case with only four parameters having two or three values each, the reduction for pairwise is up to 75%. In another case with 75 parameters, the reduction was much higher since only 28 test sets were needed to cover all pairwise combinations of parameters, while exhaustively a tester needed to create 10^{29} test sets. The authors also mention that one of the developers uncovered all faults with pairwise that he had previously uncovered with exhaustive unit testing. We have considered the results presented by this paper and decided to implement a combinatorial design method in this thesis. As the authors suggested, the pairwise method is one of the most used combinatorial methods and it is sufficient for revealing most of the faults of a system.

In a paper written by Lei et al., (Lei et al., 2002) the authors provide two algorithms for generating pairwise test cases: an algorithm for horizontal growth and an algorithm for vertical growth. The In-Parameter-Order strategy, as they call it, starts by pairing up the first two parameters of a system and creates a new test set for each pair. The algorithm for horizontal growth is used to extend the existing test cases with the values of a new parameter until all the pairs between the new parameter and each of the previous parameters have been satisfied. In case pairs are still left uncovered after filling in a value for each existing test set, new test sets need to be added. This is where the algorithm for vertical growth is used. It will add new test sets and cover all missing pairs. They implemented the algorithm as a tool called PairTest and compared its test generating abilities with AETG, a pairwise tool developed at Bellcore (Cohen et al., 1996). They did not find a significant difference in the number of test sets created by both tools in order to cover all the pairs of different size systems, but they found that PairTest has a lower time complexity than AETG. In Lei et al.'s paper there are two limitations. First, with respect to the horizontal algorithm, a value is added horizontally based on the one that will satisfy the most uncovered pairs. The paper does not discuss what should be done if all potential values will satisfy the same number of pairs. Second, the paper does not consider constraints. These are limitations that we have improved in our algorithm, as described in section 5.3.

Czerwonka discusses an algorithm for n-wise (Czerwonka refers to it as t-wise) testing called PICT with the aim of presenting an approach to n-wise testing that can be used in a real world setting (Czerwonka, 2008). Czerwonka states that literature exists on 20 tools for pairwise generation of test cases but most lack features necessary for them to be used within industry. It is also stated that a lot of work has been done previously to improve the time for creating pairwise arrays. The three main principles of PICT are speed of test generation, ease of use and extensibility. PICT takes parameters each containing values as input, similar to how we take elements with values. PICT is divided into two distinct steps. The first is a preparation step which builds all possible pairs and marks them as uncovered, covered or excluded. The excluded pairs are those which violate constraints. Our algorithm considers pairs that are invalid in a row during the second pair processing step. These pairs can be invalid due to other values in the row, e.g. a triplet. PICT handles this in the first step. For example if PICT is using pairwise and creates all pairs for three parameters A, B and C, but an invalid triple can exist, then PICT will also add a triplet ‘ABC’ during this initial phase and mark it as excluded. Another difference here to our approach is that we do not build all pair combinations at the start. This is due to the memory requirements of storing all of this data. Instead we build pair combinations as each element is being processed and remove the pairs once they are deemed invalid or have been covered within a test case. The second step is to process the pairs and build the array of n-wise test cases. This occurs in a similar manner to our approach by trying to cover as many uncovered pairs as possible when adding to a row. The paper also discusses the concept of mixing the n value. For example if it is known that faults are more likely to occur in one subsection between triplets of parameters then set n to 3 while processing this subsection, otherwise use 2. A related concept presented is to split the system under test (SUT) into a hierarchy applying n-wise to a level of the hierarchy and combining the result with other levels of the hierarchy. This is useful in domains with a clear hierarchy e.g. UI windows that have dialogues. This reduces combinatorial explosion. Czerwonka also discusses the ability to seed the generator with a combination of parameters with the aim of defining a common parameter set that can be reused in testing. Czerwonka also looks into negative testing by allowing one invalid value per test case to test how the SUT handles this. PICT allows for weighting parameters to put more emphasis on parameters that the tester deems more important. As our pairwise generator works in an industrial setting we have achieved this goal also set out by PICT.

Orthogonal and Covering Arrays

Williams discusses a tool called TConfig used for pairwise testing (Williams, 2000). This tool is based on a method using orthogonal and covering arrays. This approach is rooted in statistical experimental design. Williams states that empirical studies have shown that covering pairs in the test cases provides excellent code coverage. The author describes the algorithm used for pairwise. An orthogonal array is one where all pairs are covered the same number of times. It is a construct used in statistical experimentation and is defined as O(c,k,n,t) where c is the number of test cases, k the number of elements, n the values with constraints applied and t the strength e.g. for pairwise t has a value of 2. Covering arrays are defined in the same manner except that in covering arrays all pairs are covered at least once but they do not have to occur the same number of times. The proposed tool called TConfig is compared with the PairTest tool described in Lei et al.'s paper. It created less test configurations in 13 out of 16 comparisons and was, at worst, over 100 times faster than IPO. The example used did not contain any constraints and it was unclear how constraints like those we encountered could be incorporated into the algorithm. Williams also highlights that further work is needed in applying this approach for parameters with differing numbers of values. The suggestion is to merge the approach presented with the all pairs approach. As a result and due to the closer match of the All Pairs algorithms we decided on using an All Pairs approach in this thesis.

Colbourn et al. also use orthogonal and covering arrays in their pairwise algorithm, called DDA (Deterministic Density Algorithm) (Colbourn et al., 2004). It is deterministic, is fast at building test cases and gives a logarithmic worst-case guarantee on test case size. It can also be seeded with desired tests. It is compared with a number of other pairwise algorithms. As mentioned in the previous paper there is an issue with processing parameters that have an unmatched number of values. This paper suggests the use of mixed level covering arrays to solve this problem but admits that there are few results on certain aspects when applying this to software testing. They do however incorporate mixed level covering arrays into DDA. The paper mentions the need for a pairwise generator that can handle constraints but it is unclear whether DDA provides a solution for this when the DDA algorithm is presented. As a result the all pairs approach is still a better fit based on our generation requirements.

7.3. Random Generation

This section contains related work that exists in the area of software testing. The concepts they present can be applied to the random generator and present a number of interesting ideas for future work.

Visser et al. (Visser et al., 2006) discuss automated test input generation of object orientated container classes that use state matching to avoid generation of redundant tests. The input to the tests are sequences of calls to methods in the containers interface. They use exhaustive techniques with model checking to explore all possible test sequences (with a limit on the input size). They also use lossy techniques where they create abstractions of the model and use those to drive the test. This reduces the number of states being tested. The state of the test is examined to see if it can be matched with a previously stored state allowing them to avoid using this state again. They use a purely random technique as a basis for comparison in terms of coverage. These state based techniques serve as an improvement to random generation. In our case the exhaustive state space of possible configurations can be so large as to render the probability of our random generator producing duplicate configurations too small to warrant introducing state matching to our random generator. It would be an interesting future work to explore the different techniques discussed by Visser et al. in more detail.

Pacheco et al. (Pacheco et al., 2005) discuss automatic generation and classification of test inputs. The authors use a tool called Eclat whose output is a set of unit tests for the Java classes given as an input. The authors present one technique for test input selection and two additional techniques that complement the test input selection. For the first technique, which can be considered an error-detection technique, they find a small set of inputs likely to produce faults from a larger list. This is done by comparing the behaviour of the Java classes when tested against a correct model. If the input given violates this correct model, the technique classifies this as a test input likely to reveal a fault. Another component of this first technique is used to dismiss the redundant test inputs. The second technique is used to convert the test inputs of the first technique into a set of failing test cases. The third technique complies to a model based approach and is used to create valid inputs that are going to be used in the first technique. The authors have implemented these techniques in the Eclat tool and found that the final product was effective at creating test inputs likely to reveal faults. Related to our study, the valid configurations have been created by using an approach similar to the third technique mentioned in this paper. We believe that the first technique could be followed in the future work of our study when considered along with Visser et al.'s paper, since it would be interesting if the random generator could discard duplicate or redundant configurations and reduce the number of configurations to a smaller set that is likely to produce a crash.

In 2007, Pacheco et al. (Pacheco et al., 2007) discussed feedback-directed random test generation that uses the feedback received after running different randomly generated unit tests for object-oriented programs. Their technique generates inputs by randomly selecting a method call and finding elements from previously generated inputs. When an input is built, it is executed and checked against a set of contracts and filters. This helps determining if the input is illegal, contract-violating, redundant or if it can be used for generating more inputs. The three default filters used are called equality, null and exceptions. The equality filter maintains a set of all created inputs and checks each new input for equality against this set. This results in discarding redundant inputs. The null filter uncovers the absence of a null check on the arguments of a method. The exceptions filter finds an input as contract-violating if its execution throws an exception. This technique, which was implemented in a tool called RANDOOP, has proven advantages in terms of coverage and error detection when compared with systematic generation techniques or undirected random test generation techniques. It retained the advantages of the random testing technique (scalability and simplicity of implementation) and cancelled its disadvantages by avoiding the generation of redundant or not useful inputs. When the authors disabled the use of filters and contracts, their tool revealed less errors and skipped some very important bugs that were found previously when using the feedback-directed technique. The authors believe that a possible combination of both random and systematic approaches could represent a useful technique that retains the best of each approach. Feedback-directed generation techniques could also be used in the future work of our study and improve the random generator so that it can discard invalid, redundant or constraint violating configurations.

Korel discusses the differences between random, path-oriented and dynamic test data generators (Korel, 1996). In this paper, the author presents an approach for automated generation of test data for programs with procedures. His approach analyses data dependence to guide the test generation process. He mentions that random test data generators are usually inefficient at finding test data that can execute a selected statement, mainly because their random nature gives them the possibility to always miss something. The main idea of the dynamic approach is to initially execute a program with random selected inputs and during the execution of the program, to decide whether the execution should continue through the current or an alternate branch. The dynamic approach concentrates only on branches that are related to the execution of the selected statement and ignores the others. To identify these branches, this approach uses data dependencies that guide the search process. The author refers to the dynamic approach that is used in this paper as the chaining approach which identifies a series of statements to be executed before reaching the selected statement. The results of the experiment conducted by Korel indicate that the chaining approach increases the chances of generating test data. This paper has revealed some of the pitfalls of our random generator. The random generator might always choose to skip creating an optional Configuration Element and this can lead to not finding a crash that is related to the use of this Configuration Element. This is one reason why we chose to implement our solution using not only the random technique but also a more dynamic approach, the pairwise generation technique.

8. Conclusion and Future Work

Mecel AB work with automotive software. One aspect of what they do is to create AUTOSAR configurations for Basic Software modules. These configurations are provided as input to Source Code Generators which generate executable code for the modules. The focus of our thesis was to design and develop a random configuration generator and a pairwise configuration generator and compare them using a number of experiments and analyse the results. We also developed a constraints framework. Some modules have rules that must be followed in order to create a working configuration. Our framework provides a means to uncover constraint violations and provide a fix for the violation. We have implemented the constraints for the FiM module within this framework.

We focused on three experiments when comparing the two generators. In each of the experiments we used three modules, a small module called Det, a medium module called FiM and a large module called Dem. Each module has a SCG and the SCG has two key components. The first is a constraints checker that ensures the configuration conforms to the rules required of the module. The second component takes the configurations that have passed the constraints checker successfully and generates the code. Our aim was to test both of these components to determine whether the pairwise generator is at least as good as the random generator with respect to finding crashes and bugs.

The first experiment was to see how many crashes the pairwise generator could produce from the SCG for a given module and how many crashes the random generator could produce when allowed to create the same number of configurations as the pairwise generator for the same module. In this experiment no crash information was found for Det. As Det is such a small module this is most likely due to the absence of crash causing issues in the Det SCG. Both generators uncovered a bug in the second module FiM. The FiM module had some constraints it needed to conform to. As a result we handled these constraints in our constraints handler according to the specification provided by Mecel AB. We then analysed the generation reports for the SCG and found that despite giving the SCG a working configuration it threw errors for the area we had constrained. Mecel AB confirmed that the SCG contained a bug and have raised a report to have it fixed. The Dem module has a large number of constraints which we did not apply in the constraints handler but has been identified as future work and for this module neither the random generator nor the pairwise generator produced crashes. We also looked at the element coverage achieved by the random generator and pairwise generator for the three modules. In both cases the coverage was 100%.

The second experiment was similar to the first experiment except instead of limiting the random generator to a set number of configurations we allowed it to run for a set amount of time. This was to allow for the fact that the random generator has no restriction on the number of configurations it generates. Using the same three modules the results were comparable to the first experiment. No crashes were produced by either generator for Det, the same bug as in the first experiment was identified in FiM by both the pairwise generator and the random generator and no crashes were identified for Dem.

There are a number of conclusions we draw from the experimentation and other factors that carry weight, when comparing the random generator and the pairwise generator. Both generators showed the ability to uncover problems within the SCGs in the first two experiments for FiM. This finding is very positive for pairwise as it shows that the pairwise generator has been successful in an industrial setting. Czerwonka points out when discussing PICT (Czerwonka, 2008) that literature exists for 20 different pairwise tools but most of those tools do not work in an industrial setting. The pairwise generator also works in the presence of constraints, an area on which literature has not covered in great detail.

The final experiment was to see if the overhead introduced by pairwise in processing all pairs and creating all configurations for those pairs was noteworthy when compared with the time the random generator takes to create the same number of configurations for the same module(s) when considering the time taken to run the SCG for the configurations. For the Det and FiM module, we saw that the overhead introduced by using the pairwise generator was not noteworthy. The overhead for the Dem module appears to be noteworthy, but the time spent running the SCG was reduced, as only the constraints checker component ran.

There is some qualitative data that works in favour of the pairwise generator. An important aspect for industry was that the time taken to run the pairwise generator needed to be feasible enough to allow for running a large module on a nightly builder. The pairwise generator was built with this in mind and the large module Dem runs on the nightly builder in Mecel AB and finishes in the allotted time. An important aspect to consider when looking at the run time for the pairwise generator is that it only needs to be run once for a given input. It is deterministic so will always produce the same output. This means that for any changes made to a SCG the configurations that were produced by the pairwise generator can be reused without having to rerun the pairwise generator. The same cannot be said of the random generator as, due to its random nature it does not have a definition of done.

In summary the pairwise generator has made a positive contribution to the field as it is an example of a pairwise technique that uses constraints that has been shown to work within industry. On the surface it would appear that the random generator is more efficient than the pairwise generator but given that the pairwise generator only needs to run once for a given input this indicates that the initial overhead can be offset by the amount of times a tester reuses the configurations. Going forward, if the interfaces that both generators use for processing the inputs changed, there would be no real difference between what would need to change in the implementation of both algorithms. The pairwise algorithm creates the Definition Element Table using the same principle as the random generator, so we cannot assume that one generator would be more complex to update than the other. The pairwise generator is recommended especially when the number of exhaustive test cases is practically infeasible to create, which is the case for the majority of the BSW modules. For small modules, the random generator is comparable and it performs quite well, according to the results of our experiments.

Some questions arose about the various different approaches that can be taken when using pairwise. The pairwise technique is actually one flavour of the more general n-wise technique. Future work in this area of research could look into the use of n-wise where n is higher than 2, applied directly to the automotive industry. It would be interesting to discover which n value produces the best results within this particular domain.

Another idea for future work is to build a more comprehensive constraints handler for the modules. This could make use of the dependency matrix discussed in the software product lines literature. We found very promising results when testing with a module that used the constraints handler so the SCGs for more modules can be tested when more constraints are considered. It is also interesting from the perspective that it might lead to an alternate approach to the all pairs algorithm resulting in an algorithm that handles constraints differently to our approach. Mecel AB agree that this is an area of interest to them and have started discussions with future thesis students to continue with this subject.

Another area where future work could be carried out is further optimisation of the pairwise generator.

The results from this thesis should be applicable to other domains and an interesting future work would be to recreate this thesis within a different domain.

There are a number of future work ideas that could be applied to the random generator. There are interesting concepts in the use of state based random generator. States can be used to avoid duplicate constraints, to take a previously correct state and modify it slightly with the aim of producing a crash causing state and adding the ability to shrink states to the smallest possible state still likely to produce a crash. States can also be used with a tree based generator that backtracks when building a tree to try and create a unique configuration that contains elements less covered by previously created trees.

There are some limitations we have identified within this thesis. The pairwise approach we used considers pairs of 2 elements. The n-wise approach can be used for different n values, e.g. by comparing 3-wise. The reason we chose to go with pairs is due to the literature highlighting that it is between pairs of elements that faults are most likely to occur (Cohen et al., 1996). As our thesis was conducted as a case study there is a risk that a higher value of n would better produce crashes for this case.

The use of the all pairs approach to pairwise over orthogonal and covering arrays may be considered a limitation. Given the lack of information on using constraints with orthogonal and covering arrays the all pairs approach was a better match for our requirements.

Another possible limitation is that without a comprehensive constraints framework we were limited in the number of modules we could test and analyse the results for. We did manage to produce some favourable results and show that pairwise generation is at least as good as random generation. With more data it may become more definitive that one generation technique is better than the other in this domain.

A final limitation is that it was not possible to verify the source code generated by the SCGs for our configurations. For this study, of the unit tests created by Mecel AB there was no unit test suite for verifying the source code.

9. References

AUTOSAR. (2012). Basics. *AUTOSAR*. 2013, 12 March.
http://autosar.org/index.php?p=1&up=1&uup=0

AUTOSAR. (2011, December). Diagnostic Event Manager V4.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 20 May.
http://autosar.org/download/R4.0/AUTOSAR_SWS_DiagnosticEventManager.pdf

AUTOSAR. (2011, November). Function Inhibition Manager V2.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 20 May.
http://autosar.org/download/R4.0/AUTOSAR_SWS_FunctionInhibitionManager.pdf

AUTOSAR. (2012). Home. *AUTOSAR*. 2013, 28 February. http://autosar.org/index.php

AUTOSAR. (2011, October). Layered software architecture V3.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 12 March.
http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

AUTOSAR. (2012). Motivation and goals. *AUTOSAR*. 2013, 28 February.
http://autosar.org/index.php?p=1&up=1&uup=2&uuup=0

AUTOSAR. (2011, September). Specification of Development Error Tracer V3.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 21 April.
http://www.autosar.org/download/R4.0/AUTOSAR_SWS_DevelopmentErrorTracer.pdf

AUTOSAR. (2011, November). Specification of ECU configuration V3.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 14 February.
http://autosar.org/download/R4.0/AUTOSAR_TPS_ECUConfiguration.pdf

AUTOSAR. (2008, February). Technical Overview V2.2.1 R3.0 Rev 0001. *AUTOSAR . 2013,* 11 May. http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf

AUTOSAR. (2011, October). Virtual Functional Bus V2.2.0 R4.0 Rev 3. *AUTOSAR*. 2013, 13 April. http://www.autosar.org/download/R4.0/AUTOSAR_EXP_VFB.pdf

Claessen, K., & Hughes, J. (2000, September). QuickCheck: a lightweight tool for random testing of Haskell programs. In *Acm sigplan notices* (Vol. 35, No. 9, pp. 268-279). ACM.

Cohen, D. M., Dalal, S. R., Parelius, J., & Patton, G. C. (1996). The combinatorial design approach to automatic test generation. *Software, IEEE,13*(5), 83-88.

Colbourn, C. J., Cohen, M. B., & Turban, R. C. (2004, February). A deterministic density algorithm for pairwise interaction coverage. In *Software Engineering*. ACTA Press.

Czerwonka, J. (2008). Pairwise testing in the real world: Practical extensions to test-case scenarios. *Microsoft Corporation, Software Testing Technical Articles*.

Honekamp, U. (2009). The Autosar XML Schema and Its Relevance for Autosar Tools. *Software, IEEE*, *26*(4), 73-76.

Korel, B. (1996, May). Automated test data generation for programs with procedures. In *ACM SIGSOFT Software Engineering Notes* (Vol. 21, No. 3, pp. 209-215). ACM.

Lei, Y., & Tai, K. C. (2002, January). A test generation strategy for pairwise testing. In *IEEE Transactions on Software Engineering,* Vol. 28, No. 1. IEEE.

McGregor, J. (2010). Testing a software product line. *Testing Techniques in Software Engineering*, 104-140.

Mecel. (2013). Product brief – Mecel PICEA Suite. *At the forefront of automotive technology - Mecel AB*. 2013, 12 March. http://www.mecel.se/products/mecel-picea/Product.Brief.Mecel.Picea.pdf

Oriat, C. (2005). Jartege: a tool for random generation of unit tests for java classes. *Quality of Software Architectures and Software Quality*, 242-256.

Pacheco, C., & Ernst, M. (2005). Eclat: Automatic generation and classification of test inputs. *ECOOP 2005-Object-Oriented Programming*, 734-734.

Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007, May). Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on* (pp. 75-84). IEEE.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131-164.

Visser, W., Păsăreanu, C. S., & Pelánek, R. (2006, July). Test input generation for java containers using state matching. In *Proceedings of the 2006 international symposium on Software testing and analysis* (pp. 37-48). ACM.

Williams, A. W. (2000, August). Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6* (Vol. 1, pp. 59-74).

Xeger. (2009, October). A Java library for generating random text from regular expressions Version 1.0-SNAPSHOT. Google Project Hosting. 2013, 02 June. https://code.google.com/p/xeger/

Ye, H., & Liu, H. (2005, June). Approach to modelling feature variability and dependencies in software product lines. In *Software, IEE Proceedings-* (Vol. 152, No. 3, pp. 101-109). IET.

Yin R. K. (2009). Case Study Research: Design and Methods. *SAGE Publications, USA.* (4th edition,Vol.5, pp. 5-35).

Appendices

Appendix A: Det module Parameter Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0">
 <AR-PACKAGES>
  <AR-PACKAGE UUID="a3e46e8f-d03a-4bfd-8944-d551e20f8711">
   <SHORT-NAME>PICEA</SHORT-NAME>
   <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>BswMd</SHORT-NAME>
      <ELEMENTS>
    <BSW-IMPLEMENTATION UUID="540c3526-be04-4b34-b6c8-3e593eb0e5b0">
     <SHORT-NAME>BSWImpl_Det</SHORT-NAME>
     <VARIATION-POINT/>
     <PROGRAMMING-LANGUAGE>C</PROGRAMMING-LANGUAGE>
     <SW-VERSION>3.0.1</SW-VERSION>
     <VENDOR-ID>41</VENDOR-ID>
     <AR-RELEASE-VERSION>4.0.3</AR-RELEASE-VERSION>
     <BEHAVIOR-REF DEST="BSW-INTERNAL-
BEHAVIOR">/PICEA/BswMd/Det/BSWBehavior_Det</BEHAVIOR-REF>
    </BSW-IMPLEMENTATION>
    <BSW-MODULE-DESCRIPTION>
     <SHORT-NAME>Det</SHORT-NAME>
     <LONG-NAME>
      <L-4 L="FOR-ALL">Development Error Tracer</L-4>
     </LONG-NAME>
     <MODULE-ID>15</MODULE-ID>
     <INTERNAL-BEHAVIORS>
      <BSW-INTERNAL-BEHAVIOR>
       <SHORT-NAME>BSWBehavior_Det</SHORT-NAME>
       <EXCLUSIVE-AREAS/>
       <ENTITYS/>
       <EVENTS/>
      </BSW-INTERNAL-BEHAVIOR>
     </INTERNAL-BEHAVIORS>
    </BSW-MODULE-DESCRIPTION>
      </ELEMENTS>
     </AR-PACKAGE>
    <AR-PACKAGE UUID="ef01f2f3-4873-4ec4-8c78-9c9462b57a02">
     <SHORT-NAME>EcucDef</SHORT-NAME>
     <ELEMENTS>
      <ECUC-MODULE-DEF UUID="9156748c-7fe3-45b6-8264-c1e7d3b77faa">
       <SHORT-NAME>Det</SHORT-NAME>
       <DESC>
        <L-2 L="EN">Det configuration includes the functions to be called at notification. On one</L-2>
       </DESC>
       <ADMIN-DATA>
        <DOC-REVISIONS>
         <DOC-REVISION>
          <REVISION-LABEL>4.2.0</REVISION-LABEL>
          <ISSUED-BY>AUTOSAR</ISSUED-BY>
          <DATE>2011-11-09</DATE>
         </DOC-REVISION>
        </DOC-REVISIONS>
       </ADMIN-DATA>
       <INTRODUCTION>
        <P>
         <L-1 L="EN">side the application functions are specified and in general it can be decided
           whether Dlt shall be called at each call of Det.</L-1>
        </P>
       </INTRODUCTION>
       <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
       <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
      <REFINED-MODULE-DEF-REF DEST="ECUC-MODULE-DEF">/AUTOSAR/EcucDefs/Det</REFINED-
  MODULE-DEF-REF>
       <SUPPORTED-CONFIG-VARIANTS>
        <SUPPORTED-CONFIG-VARIANT>VARIANT-PRE-COMPILE</SUPPORTED-CONFIG-VARIANT>
       </SUPPORTED-CONFIG-VARIANTS>
       <CONTAINERS>
        <ECUC-PARAM-CONF-CONTAINER-DEF UUID="5aa195a3-1a24-4c92-9046-43ee0cb9d847">
         <SHORT-NAME>General</SHORT-NAME>
         <DESC>
          <L-2 L="EN">Generic configuration parameters of the Det module.</L-2>
         </DESC>
         <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
         <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
         <MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
         <PARAMETERS>
          <ECUC-BOOLEAN-PARAM-DEF UUID="5b338520-29ef-4ce8-acbf-e040aab3f534">
           <SHORT-NAME>Dlt</SHORT-NAME>
           <DESC>
           <L-2 L="EN">
            Only if the parameter is present and set to true, the Det requires the Dlt interface and forwards its
            call to the function Dlt_DetForwardErrorTrace. In this case the optional interface to Dlt_Det is required.
           </L-2>
           </DESC>
           <LOWER-MULTIPLICITY>0</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><DEFAULT-VALUE>FALSE</DEFAULT-
  VALUE>
          </ECUC-BOOLEAN-PARAM-DEF>
          <ECUC-BOOLEAN-PARAM-DEF UUID="3448ae28-ee36-4923-b4e1-16d21cb09e23">
           <SHORT-NAME>VersionAPI</SHORT-NAME>
           <DESC>
           <L-2 L="EN">Pre-processor switch to enable / disable the API to read out the modules version information.
           </L-2>
           </DESC>
           <INTRODUCTION>
           <P>
            <L-1 L="EN">
             true: Version info API enabled.
             false: Version info API disabled.
            </L-1>
           </P>
           </INTRODUCTION>
           <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><DEFAULT-VALUE>TRUE</DEFAULT-
  VALUE>
          </ECUC-BOOLEAN-PARAM-DEF>
          <ECUC-ENUMERATION-PARAM-DEF UUID="e217ada2-44aa-47c4-897c-eb225df33d49">
           <SHORT-NAME>Platform</SHORT-NAME>
           <CONFIGURATION-CLASS-AFFECTION/>
           <ORIGIN>Mecel</ORIGIN>
           <DEFAULT-VALUE>WIN32</DEFAULT-VALUE>
```

```
           <LITERALS>
            <ECUC-ENUMERATION-LITERAL-DEF UUID="86253bdc-b9d4-4a52-b3d5-84a2b9b6d4bb">
             <SHORT-NAME>WIN32</SHORT-NAME>
            </ECUC-ENUMERATION-LITERAL-DEF>
            <ECUC-ENUMERATION-LITERAL-DEF UUID="e66b91f4-8f9e-4967-9db6-4f5cba60c957">
             <SHORT-NAME>VAST</SHORT-NAME>
            </ECUC-ENUMERATION-LITERAL-DEF>
           </LITERALS>
          </ECUC-ENUMERATION-PARAM-DEF>
          <ECUC-BOOLEAN-PARAM-DEF UUID="6f2af97c-6270-405f-a0dd-4c3f7b4570de">
           <SHORT-NAME>ForeignModule</SHORT-NAME>
           <ORIGIN>Mecel</ORIGIN><DEFAULT-VALUE>TRUE</DEFAULT-VALUE>
          </ECUC-BOOLEAN-PARAM-DEF>
         </PARAMETERS>
        </ECUC-PARAM-CONF-CONTAINER-DEF>
        <ECUC-PARAM-CONF-CONTAINER-DEF UUID="e4daed99-1b51-4cdf-b61e-816633992b67">
         <SHORT-NAME>Notification</SHORT-NAME>
         <DESC>
          <L-2 L="EN">Configuration of the notification functions.</L-2>
         </DESC>
         <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
         <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
         <MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
         <PARAMETERS>
          <ECUC-FUNCTION-NAME-DEF UUID="0af9fc8a-70f8-4996-b639-a31e8922522b">
           <SHORT-NAME>ErrorHook</SHORT-NAME>
           <DESC>
            <L-2 L="EN">
             Optional list of functions to be called by the Development Error Tracer in context of each call of
Det_ReportError.
            </L-2>
           </DESC>
           <INTRODUCTION>
            <P>
             <L-1 L="EN">
              The type of these functions shall be identical the type of Det_ReportError itself:
              Std_ReturnType (*f)(uint16, uint8, uint8, uint8).
             </L-1>
            </P>
           </INTRODUCTION>
           <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
           <UPPER-MULTIPLICITY>2</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-FUNCTION-NAME-DEF-VARIANTS>
            <ECUC-FUNCTION-NAME-DEF-CONDITIONAL/>
           </ECUC-FUNCTION-NAME-DEF-VARIANTS>
          </ECUC-FUNCTION-NAME-DEF>
         </PARAMETERS>
        </ECUC-PARAM-CONF-CONTAINER-DEF>
       </CONTAINERS>
      </ECUC-MODULE-DEF>
     </ELEMENTS>
    </AR-PACKAGE>
   </AR-PACKAGES>
  </AR-PACKAGE>
 </AR-PACKAGES>
</AUTOSAR>
```

Appendix B: Experiment Data

- Number of crashes based on set number of configurations
  - Det module

Nr.	Configurations	Crashes from Random	Crashes from Pairwise	Errors from Random	Errors from Pairwise
1	11	0	0	0	0
2	11	0	0	0	0
3	11	0	0	0	0
4	11	0	0	0	0
5	11	0	0	0	0
6	11	0	0	0	0
7	11	0	0	0	0
8	11	0	0	0	0
9	11	0	0	0	0
10	11	0	0	0	0

  - FiM module

Nr.	Configurations	Crashes from Random	Crashes from Pairwise	Errors from Random	Errors from Pairwise
1	504	0	0	22	35

2	504	0	0	45	35
3	504	0	0	53	35
4	504	0	0	36	35
5	504	0	0	61	35
6	504	0	0	39	35
7	504	0	0	50	35
8	504	0	0	36	35
9	504	0	0	48	35
10	504	0	0	32	35

- Dem module

Nr.	Configurations	Crashes from Random	Crashes from Pairwise	Errors from Random	Errors from Pairwise
1	2431	0	0	2431	2431
2	2431	0	0	2431	2431
3	2431	0	0	2431	2431
4	2431	0	0	2431	2431
5	2431	0	0	2431	2431
6	2431	0	0	2431	2431
7	2431	0	0	2431	2431

8	2431	0	0	2431	2431
9	2431	0	0	2431	2431
10	2431	0	0	2431	2431

- Number of crashes based on set amount of time
    - Det module

Nr.	**Pairwise Config.**	**Random Config.**	**Crashes from Pairwise**	**Crashes from Random**	**Errors from Pairwise**	**Errors from Random**	**Time**
1	11	3257	0	0	0	0	1h
2	11	3455	0	0	0	0	1h
3	11	3229	0	0	0	0	1h
4	11	3245	0	0	0	0	1h
5	11	3202	0	0	0	0	1h
6	11	3173	0	0	0	0	1h
7	11	3458	0	0	0	0	1h
8	11	3181	0	0	0	0	1h
9	11	599	0	0	0	0	1h
10	11	1333	0	0	0	0	1h

    - FiM module

Nr.	Pairwise Config.	Random Config.	Crashes from Pairwise	Crashes from Random	Errors from Pairwise	Errors from Random	Time
1	504	6079	0	0	35	671	2h
2	504	6088	0	0	35	604	2h
3	504	6316	0	0	35	611	2h
4	504	6097	0	0	35	596	2h
5	504	6024	0	0	35	582	2h
6	504	5974	0	0	35	576	2h
7	504	6319	0	0	35	608	2h
8	504	5968	0	0	35	576	2h
9	504	1064	0	0	35	101	2h
10	504	1958	0	0	35	194	2h

- Dem module

Nr.	Pairwise Config.	Random Config.	Crashes from Pairwise	Crashes from Random	Errors from Pairwise	Errors from Random	Time
1	2431	10680	0	0	2431	10680	4h
2	2431	10538	0	0	2431	10538	4h
3	2431	10521	0	0	2431	10521	4h
4	2431	10593	0	0	2431	10593	4h

5	2431	10819	0	0	2431	10819	4h
6	2431	10806	0	0	2431	10806	4h
7	2431	10366	0	0	2431	10366	4h
8	2431	10834	0	0	2431	10834	4h
9	2431	10881	0	0	2431	10881	4h
10	2431	10823	0	0	2431	10823	4h

- Overhead of time taken to generate configurations
    - Det module

Nr.	**A:Pairwise conf. time (s)**	**B:Random conf. time (s)**	**C:Pairwise conf. & SCG time (s)**	**D:Random conf. & SCG time (s)**
1	0.33	0.06	13.2	12.53
2	0.34	0.08	12.98	12.6
3	0.31	0.08	13.07	12.56
4	0.31	0.06	13.04	12.54
5	0.3	0.08	12.93	12.57
6	0.34	0.08	13.15	12.54
7	0.28	0.08	13.02	12.56
8	0.3	0.06	12.95	12.56
9	0.28	0.06	13.06	12.54

10	0.28	0.06	12.9	12.7

- FiM module

Nr.	A:Pairwise conf. time (s)	B:Random conf. time (s)	C:Pairwise conf. & SCG time (s)	D:Random conf. & SCG time (s)
1	23.09	2.76	583.52	560.9
2	22.95	2.96	616.79	594.19
3	23.29	3.07	617.36	592.43
4	22.42	2.82	613.6	590.29
5	23.31	2.95	624.36	602.41
6	23.38	2.96	628.37	604.16
7	22.5	2.82	587.08	562.12
8	23.29	2.92	629.02	603.5
9	121.14	20.22	3530.52	3405.78
10	65.29	12.51	1917.57	1856.29

- Dem module

Nr.	A:Pairwise conf. time (s)	B:Random conf. time (s)	C:Pairwise conf. & SCG time (s)	D:Random conf. & SCG time (s)
1	6880.31	82.01	9808.35	3120.08
2	7795.83	71.24	10912.92	3276.9

3	6876.82	78.3	9854.32	3272.8
4	7933.01	85.77	11020.66	3455.67
5	7211.48	55.68	10243.26	3057.33
6	6902.39	58.95	9942.07	3064.72
7	6897.88	80.79	9857.84	3295.29
8	8170.67	57.64	11215.2	3080.75
9	26311.54	55.74	29474.7	3050.64
10	17262.55	55.54	21144.3	3080.3

- Number of covered elements
    - Det module

Nr.	**Nr. of elements**	**Elements covered by Random**	**Elements covered by Pairwise**
1	7	7	7
2	7	7	7
3	7	7	7
4	7	7	7
5	7	7	7
6	7	7	7
7	7	7	7

8	7	7	7
9	7	7	7
10	7	7	7

- FiM module

Nr.	**Nr. of elements**	**Elements covered by Random**	**Elements covered by Pairwise**
1	35	35	35
2	35	35	35
3	35	35	35
4	35	35	35
5	35	35	35
6	35	35	35
7	35	35	35
8	35	35	35
9	35	35	35
10	35	35	35

- Dem module

Nr.	**Nr. of elements**	**Elements covered by Random**	**Elements covered by Pairwise**

1	195	195	195
2	195	195	195
3	195	195	195
4	195	195	195
5	195	195	195
6	195	195	195
7	195	195	195
8	195	195	195
9	195	195	195
10	195	195	195