Automatic Generation of AUTOSAR Software Component Descriptions

Study Thesis in Computer Sciences

by

Christopher Mutschler

[redacted]

written for

Institute of Computer Sciences
Department of Computer Sciences 2
Programming Systems
Friedrich-Alexander-University Erlangen-Nuremberg
(Prof. Dr. M. Philippsen)

Supervisor:
PD Dr.-Ing. Gabriella Kókai
Dipl.-Ing. (TU) Rudolf Grave (Elektrobit Automotive)

Start of work: 2008-11-15
End of work: 2009-04-07

Study Thesis

Thema: Automatische Generierung von AUTOSAR Software Component Descriptions

Hintergrund: Die Entwicklung von Software für tief eingebettete Systeme im Automobilbereich unterscheidet sich in einigen Punkten von der Entwicklung für einzelne oder größere Rechner. Im Fahrzeug kommen viele Prozessoren und Steuergeräte zum Einsatz, welche untereinander stark vernetzt sind und die aus Kostengründen jeweils möglichst mit nur so vielen Ressourcen ausgestattet sein sollen, wie erforderlich. Folglich müssen daher die Systeme des Fahrzeugs möglichst im Ganzen geplant werden. Gleichzeitig sollte aber eine gewisse Modularität einzelner Softwarekomponenten erhalten bleiben, um die Entwicklungskosten zu minimieren. Ein Konsortium von Konzernen der Automobilindustrie entwickelt daher den sogenannten AUTOSAR Standard (AUTomotive Open System ARchitecture). Um eine Applikation in ein AUTOSAR-System einbetten zu können, ist es notwendig ihre Kommunikation zu beschreiben, d.h. es muss beschrieben werden welche Dienste die Applikation anbietet und in Anspruch nimmt, welche Daten versendet oder empfangen werden. Hierfür muss ein spezielles XML-Dokument erstellt werden, die sogenannte Software Component Description (SW-C Description).

Aufgabenstellung: Für eine Applikation müssen in Folge dessen nicht nur die entwickelten C-Dateien, sondern zusätzlich die erwähnte SW-C Description (SW-CD) gewartet werden. Letzteres ist zeitaufwändig und, da viele Daten in mehreren Dokumenten redundant gehalten werden, eine potentielle Fehlerquelle. Es ist daher zu untersuchen, ob die XML-Dokumente automatisch aus den C-Dateien generiert werden können. Dazu wird es allerdings notwendig sein, zusätzliche Metainformation in den C-Dateien mitzuführen.

Es soll geprüft werden, welche Metainformationen für eine korrekte und sinnvolle SW-CD notwendig sind. Weiterhin soll der formale Rahmen entwickelt werden, um diese Metainformation in C-Dateien einzubringen. Schließlich soll ein Tool entwickelt werden, das den C-Code zusammen mit den Metainformationen einliest und daraus automatisch eine passende SCD erstellt.

Betreuung: Dipl.Ing. (TU) Rudolf Grave, PD Dr.-Ing. Gabriella Kókai

Bearbeiter: Christopher Mutschler

Abstract

Die Entwicklung von Software-Komponenten auf AUTOSAR-Basis unterscheidet sich in einigen Punkten von der Entwicklung von Software für herkömmliche Rechner. Es ist wichtig, die Komponenten vor der Integration in das Betriebssystem so zu beschreiben, dass sie mit einer Minimalzahl an benötigten Ressourcen auskommen können. Gleichzeitig sollen die Software-Komponten so modular wie möglich gehalten werden, um einen hohen Grad an Wiederverwendbarkeit zu garantieren. Diese Beschreibung muss von Software-Entwicklern parallel zum entwickeltem Quellcode gewartet und erweitert werden. Dies ist ein relativ fehleranfälliger Prozess.

Die vorliegende Studienarbeit untersucht die Möglichkeiten, nötige Informationen aus dem Quellcode auszulesen. Außerdem wird die Möglichkeit gegeben, nicht vorhandene Information als Kommentar ablegen zu können, um somit die komplette Software-Beschreibung aus der Quellcode-Datei heraus erzeugen zu können. Der somit entwickelte Parser und Codegenerator wird hierbei so generisch wie möglich gehalten, um eine möglichst einfache Erweiterbarkeit für zukünftige AUTOSAR Versionen zu ermöglichen.

The development of software components for AUTOSAR platforms differs from the development of conventional software components in various points. It is important to describe the components previously in order to minimize the resource consumption. Coincidentally the components should be written and described as modular as possible to increase reusability. This description has to be maintained and enhanced by developers concurrently. This can easily lead to inconsistent states.

The present thesis investigates the possibilities to read needed information out of the source code. Furthermore, the ability to complement lacking information by adding specific commentaries is given in order to generate an entire Software Component Description out of the given source code file. The developed parser and the code generator will be held as generic as possible to enable easy extensibility for further AUTOSAR releases.

Contents

1 Introduction

Elektrobit (EB) Automotive is member of the *AUTOSAR (AUTomotive Open System Architecture)* initiative as a premium member and therefore deeply involved in the standardization process between the car manufacturers, suppliers and tool developers.

The AUTOSAR operating system provides standardized Basic Software consisting of basic modules like memory management and diagnostic services. Built on that there is an *Application Abstraction Layer (AAL)* which enables everyone the development of software completely independent from the hardware and only dependent on the Middleware which provides the Runtime Environment (RTE) as the Application Abstraction. Software Components that will be developed in consideration of the RTE have to be described by Software Component Descriptions.

The creation of AUTOSAR Software Component Descriptions is a big issue for automotive software development. Maintaining the Software Component Description means also maintaining the source code of the component as well.

Since the maintenance of both the description and the code could easily lead to inconsistent states it would be appropriate to integrate a little more information in the source code to automatically generate its Software Component Description out of it. The elaboration shows how to implement such a generator software and points out problems and future prospectives.

1.1 Goals of the Project

As mentioned before software for *Electronic Control Units (ECUs)* has to be planed as an entire object. Application software for AUTOSAR is separated in several *Software Components (SW-C)*, which manage to serve a functionality by working together. Software Components are realized through normal C-functions and if they were designed properly, they could be reused as particular blocks on other ECUs. Hence, it is necessary to design the Software Component and to define its interfaces previously to the implementation in order to observe the modularity. Each modular Software Component possesses metainformation which contains descriptions about its functionality and communication. A fundamental element within this specification is the *Internal Behaviour* which describes the functionality and internal structure of a Software Component. The complete specification of the Software Component is called a Software Component Description and is written in XML.

To address the problem of modularity some tools for creating Software Components and its Software Component Descriptions are developed. Unfortunately, none of them connects the information of the descriptions directly with the source code itself, which means that there is no visuable link between them.

A tool that could get a connection between metainformation and source code and generate a Software Component Description would reduce costs for maintenance in many cases. The goal of this thesis is to analyze the minimal amount of metadata that is needed to generate the complete description and to implement a tool that realizes the generation process.

1.2 Requirements

In order to automatically generate code it is mandatory to observe the following requirements:

- With the fact that AUTOSAR is still in development, there is the possibility that there will be added or changed Application Programming Interface (API) functionality. To face this issue it is mandatory to design the implementation easily adaptable for another software developer.
- The generator should apply AUTOSAR 2.1 as its basis, in other words the structure of the Software Component Description output as well as the available amount of API function provided by the Runtime Environment of AUTOSAR.
- In order to increase the usability of the tool the amount of metainformation has to be kept as low as possible. If there is too much information needed, it would be easier to draw up the description otherwise.
- A documentation should be provided to enable developers to enhance the project in the future.
- The thesis shall investigate and implement only the common applied items. To implement all existing possibilities would just increase the afford of programming and should be no problem to be managed later on.
- The usage of the generator software should be as easy as possible. This means to provide a pre-compiled archieve in order to apply the generator and to provide makefiles in order to rebuild the tool quickly.
- The structure of the parser-generator software should be easy and uncomplicated. Furthermore, there should be enough tutorials and examples on the web to ensure that other developers can quickly feel comfortable with that tool.

1.3 Structure of the Thesis

This chapter introduces the reader to the main properties of the project. The next part shows some other ways to get a Software Component Description without writing XML-code. The second part introduces the reader to the basic software that is needed such as AUTOSAR, ANother Tool for Language Recognition (ANTLR) and EB tresos Studio. It mentions some facts about Software Component Descriptions and how they are used in the complete ECU generation process for automotive software. The analysis part is the third part of this thesis. It examines how Software Component Descriptions are structered and points out how to make use of redundant information. Furthermore, problems which have to be solved by using metadata in order to complete the information that is needed are described. In chapter 4 the reader gets acquainted to the design and implementation. This chapter shows how the metadata has to be constructed. Starting with the parsing process, it goes through the complete generation process including the generation of the Internal Behaviour. To validate the funcionality of the generator an evaluation is given to compare the generator to another software tool that is often used in the automotive industry. The chapter points out advantages and disadvantages of each tool. Chapter 6 sums up the results of this thesis and shows some perspectives for the future as well as possible enhancements.

2 Basics

This chapter gives a short overview of topics that are important for the thesis. It includes a short introduction to the available tools and formats. It is supposed to be a short reference, which can be utilized to understand details in several upcoming chapters.

2.1 About AUTOSAR

AUTOSAR is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. [9]

The AUTOSAR initiative is a collection of Original Equipment Manufacturers (OEM) and automotive suppliers with the goal of working together and establishing an open industry standard for automotive software development. The motivation is to reduce costs for maintenance and development by increasing the reusability of software.

On the one hand reusability of Software Components implies to hold them as modular as possible. On the other hand Software Components should allocate as few resources as possible. In order to ensure a minimal amount of hardware resources all communication partners and memory allocations have to be defined previous to the generation of the binary. This also implies that all communication points, communication types and communication partners have to be known during the development process to reduce software overhead. [AS_BAS02]

Figure 1 shows that AUTOSAR is strictly seperated in different layers. This layered architecture ensures that the software for the ECU can be developed completely hardware independent. AUTOSAR provides the Runtime Environment (RTE) that manages signal and port mapping as well as inter- and intra-ECU communication. Thus Software Components can easily be downloaded on other microcontroller hardware without the need of touching code within the application as Figure 2 shows. Indeed if the software component is moved onto another ECU it may be necessary to configure the communication stack of AUTOSAR.

The Software Components always communicate over the *Virtual Functional Bus (VFB)*, which is implemented by the RTE. The advantage is that Software Components do not need to know on which controller they are currently working. The RTE manages either to copy the transmitted dataelements directly to the receiving Software Component without invoking basic software or to call the communication stack if the data has to be transmitted to another ECU. The communication stack is an AUTOSAR defined cluster with modules that arrange the communication between ECUs. With the fact

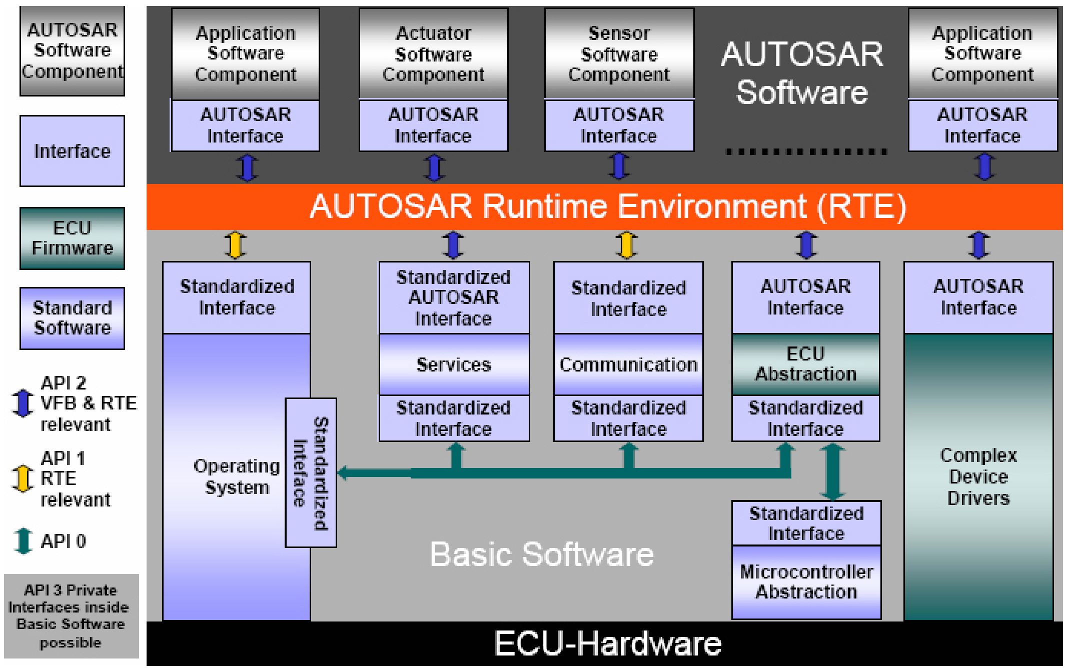
AUTOSAR Software Component
Interface
ECU Firmware
Standard Software
API 2 VFB & RTE relevant
API 1 RTE relevant
API 0
API 3 Private Interfaces inside Basic Software possible
Application Software Component
Actuator Software Component
Sensor Software Component
AUTOSAR Software
AUTOSAR Interface
AUTOSAR Runtime Environment (RTE)
Standardized Interface
Standardized AUTOSAR Interface
Services
Communication
ECU Abstraction
Operating System
Microcontroller Abstraction
Complex Device Drivers
Basic Software
ECU-Hardware

Figure 1: AUTOSAR software architecture [8]

that the RTE knows which Software Components are currently running, it is able to skip bus communication if both are running on the same ECU. The communication between Software Components has to be defined previous to the RTE generation process of course. The information about communication has to be described in the Software Component Descriptions.

2.2 EB tresos Studio

EB tresos Studio is the tool developed by Elektrobit Automotive that enables the configuration of the embedded software modules. It allows to completely configure each module of the AUTOSAR Basic Software and its developed applications at pre-compile time. Therefore it is also possible to detect configuration errors or unaccomplished dependencies between Software Components.

If the configuration is finished *tresos Studio* generates the proper code and the entire operating system including applications that has been developed. After the generation the entire collection can be compiled and linked to a binary that can be downloaded onto the target microcontroller.

EB tresos Studio provides a Software Component Description Importer that validates the formal structure of the Software Component Description. It determines whether the description is invalid or not well-formed by mapping it against the scheme definition. In

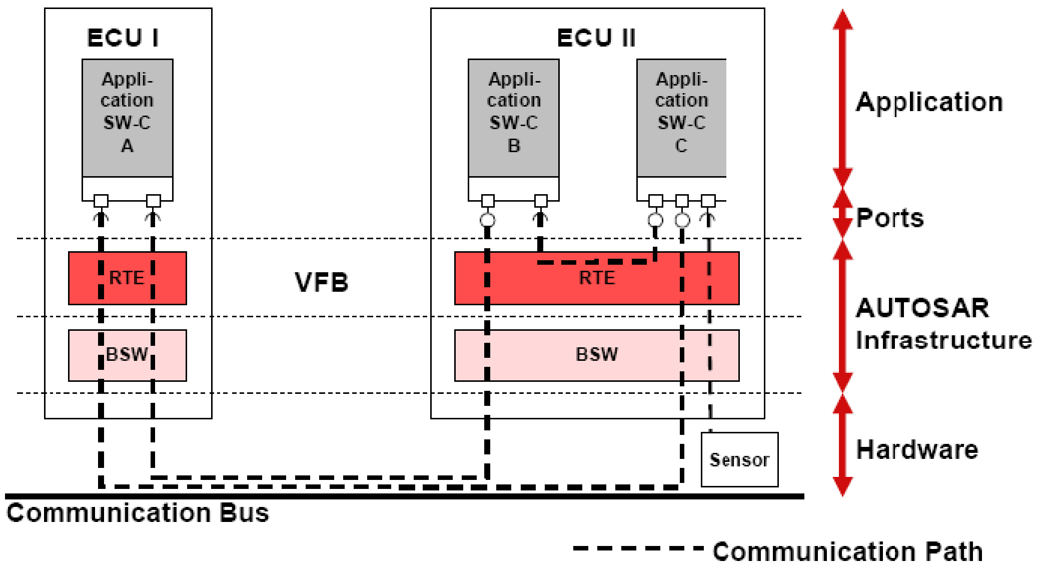

Figure 2: Interaction between Software Components on the same ECU and on different ECUs [10]

case of correct input the description will be imported and the RTE can be configured properly.

2.3 ANother Tool for Language Recognition

ANother Tool for Language Recogntion (ANTLR) is an object oriented parser generator that is being developed since 1989 at the University of San Francisco. ANTLR is not dependent on the programming language. ANTLR can either be programmed with C/C++, C#, Java or Python. For the present purpose, Java has been selected. [12]

The usage of ANTLR is relatively easy compared with other parser generators. In addition a huge amount of examples and tutorials can be found on the web so that the parser developed here can easily be enhanced.
An ANTLR source code file simply consists of four major parts [11]:

1. Token definitions: a token consists of a sequence of characters, i.e. strings.

2. Lexer (lexical scanner) rules: a lexical scanner is often organized as a seperate scanner and tokenizer functions. A lexer decides depending on the lexemes it is about to read how to tokenize the given input stream.

3. Parser rules: a parser reads a token stream that is provided by the lexer and matches phrases in the language via rules.

4. Implementation in target language: in order to perform some semantic actions for each parsed phrase it is necessary to include source code in the parser rules to invoke custom actions.

2.3.1 Grammar Specification

Example 3 shows how to set up the implementation part of the parser. Each item that belongs to the `@members` part will be completely applied to the generated source code without being touched. In this example the parser is said to open the file given as argument on the command line, to instantiate the parser and to begin with the parser rule that is called `expr()`.

```
grammar SimpleCalc;      // Name of the specified grammar

options {
  language=Java;         // programming language in which the
}                        // parser rules shall be compiled

tokens {                 // token definitions
  PLUS = '+' ;    MINUS = '-' ;
  MULT = '*' ;    DIV   = '/' ;
}

// This part will be directly copied into the generated parser file
@members {
  public static void main (String[] args) throws Exception {
    SimpleCalcLexer lex = new SimpleCalLexer(new ANTLRFileStream(args[0]);
    CommonTokenStream tokens = new CommonTokenStream(lex);
    SimpleCalcParser parser = new SimpleCalcParser(tokens);

    try {
      parser.expr();
    } catch (RecognitionException e) {
      e.printStackTrace();
}}}
```

Figure 3: Simple Calculator Implementation

In the *options* section a variety of available options can be declared. Some of them are:

- diagnostic: generates a text file with debugging information for the grammar specification

- traceParser: traces the parser rules
- traceLexer: traces the lexer rules

It is also possible to set default error handlers, selecting the target language, changing the default lookahead value and many more [1]. In addition to the previously given code two more sections are required in order to get a working parser: the parser rules and the lexer rules.

The rules can easily be written in extended Backus-Naur-Form, which is also readily to understand. Given the specification above, the parser and lexer rules can be written as follows:

```
// Parser rules

expr   :  term ( (PLUS | MINUS ) term ) *   ;
term   :  factor (  (  MULT | DIV  ) factor ) * ;
factor :  NUMBER  ;

// Lexer rules

NUMBER : (DIGIT)+ ;
WHITESPACE: ('\t' | ' ' | '\r' | '\n' | '\u000C' )+ {$channel=HIDDEN;}
fragment DIGIT : '0'..'9' ;
```

Most of the code showed here should be easy to understand but there are two little parts that need to be explained closer: the keywords `$channel` and `fragment`.

The `$channel=HIDDEN` keyword simply says that the here named `WHITESPACE`s shall be ignored/skipped during the parsing process. The `fragment` keyword is comparable with the rather known protected/private keyword from object oriented programming languages and simply means that the defined rule is not usable outside the lexer rules. For the parser rules this can also bedefined.

When the code has been written, it can be compiled and tested. Imagine the following example testcases:

```
1+2+3+4
1++1++2++3
1+4+8+
```

The first line will be parsed accurately by the parser because it matches to the given parser rules. Anyway `1++1++2++3` does not match to the given grammar and will result in the following output:

```
line 1:2 extraneous input '+' expecting NUMBER
line 1:5 extraneous input '+' expecting NUMBER
line 1:8 extraneous input '+' expecting NUMBER
```

Similar will be case number three:

```
line 0:-1 extraneous input '<EOF>' expection NUMBER
```

In respect to that a small simple calculator example can be implemented intuitively by writing a few lines of code. In addition to that a variety of existing grammars already exist for ANTLR so that a given C grammar can easily be enhanced to process the metainformation needed to solve the given problem.

2.3.2 Advantages of ANTLR

A variety of parser generators are under free license. ANTLR has, comparing to other parser generator, many advantages:

- The source code, which is written for ANTLR is easy to understand, e.g. it is intuitively clear what the parser does at each point of interest.

- Considering maintenance, which is especially done by persons who are not familiar with parser generators: ANTLR strictly separates between parser and lexer rules. Hence, it is obvious where to place which code.

- ANTLR comes with a huge and very strong community. In the web exist many tutorials, mailinglists and forums that every question will be answered as fast as possible. Thus it is ensured that prospective developers will get very good support.

All these advantages lead to the decision to choose ANTLR in order to get a working parser that implements the given grammar. Writing such a grammar in JavaCC or yacc (yet another compiler compiler) will be much harder regarding the fact that there exists nearly no reading materials. For yacc it is a matter of fact that in 2008 an OpenBSD developer fixed a bug that was over 25 years old. Beyond that the provided documentations for yacc as well as for JavaCC are not very helpful at all. On the other hand JavaCC provides an easy Java-like programming paradigm. Writing a parser is very easy regarding the fact that

```
  Token t = <TOKEN>
```

will assign the token to the variable `t`. Its string can easily be obtained by `t.image`. Nevertheless, the various points are not convincing at because ANTLR is still in development and just released in its new version 3. This effort shows the strong orientation towards ANTLR.

2.4 AUTOSAR Software Component Descriptions

The *Software Component Template* defines all kinds of elements a Software Component consists of. A Software Component structures the system functionality and is mainly divided into the following sections [7]:

- Hierarchical Structure: Software Components have to be combined to a Software Composition. Software Compositions again can be combined. Structuring the Software Components enables suppliers to combine their components to complete compositions that can be delivered as a complete package.

- Ports and Interfaces: the RTE has to know which ports are going to be used and which dataelements are transmitted. Interfaces aggregate dataelements that are provided or required by a port using this Interface.

- Internal Behaviour: the Internal Behaviour gives an overview about the structure of the Software Component. It defines Runnable Entities, Exclusive Areas and much more. Runnable Entities structure the Software Component's functionality and are executed independently from each other. The Internal Behaviour can be understood as a reflection of the source code.

- Implementation: the Implementation just gives information of the available formats. An entire Software Component can either be present as source code or as pre-compiled object code. Object code is very useful if suppliers provide their Software Components without willing to hand out the source code. Furthermore it is appointed that the code can not be changed after delivery.

A special type of Software Components are the *Atomic Software Components*. They are undividable Software Component types and contain the minimal possible functionality of a Software Component. An Atomic Software Component communicates via dedicated Ports over the VFB. The VFB is specified by AUTOSAR and a communication abstraction provided by the RTE.

As shown in Figure 4 the present way to develop automotive software is to write the description and the code separately. In most cases system design tools like *dSPACE SystemDesk* give the ability to draw the entire system with all its components in a *Graphical User Interface (GUI)*. Afterwards they check the configuration and export a Software Component Description as well as code skeletons for the Software Components.

If any time later, the application developer needs to change properties of his Software Components (which can merely be to rename a port or an interface) the generation of the code skeleton will lead to problems. The developer already wrote the code for his Software Component and gets then a new file and has to add his source code manually.

Therefore it is appropriate to change this workflow as shown in Figure 5. With this changing sequence many problems are solved. For instance if a designer decides to

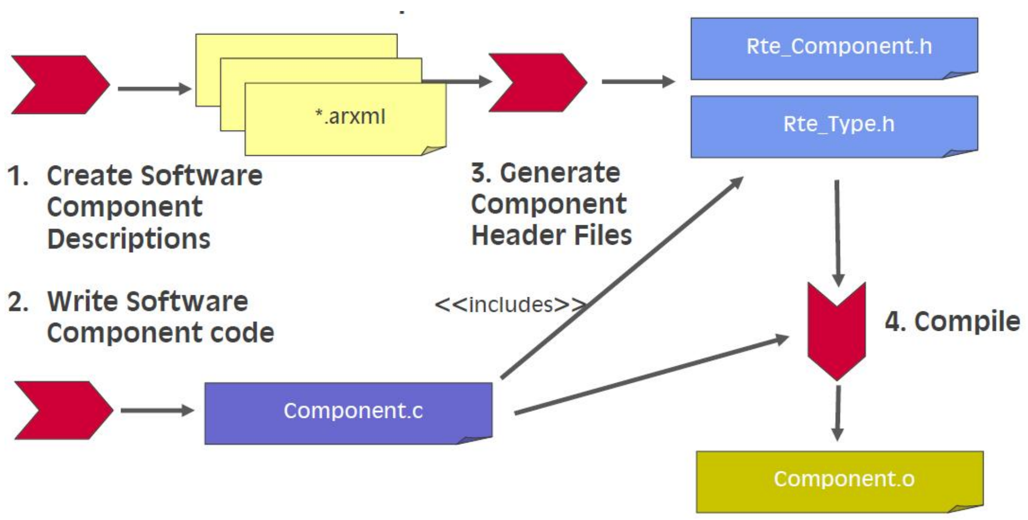
*.arxml
Rte_Component.h
Rte_Type.h
1. Create Software Component Descriptions
3. Generate Component Header Files
2. Write Software Component code
<<includes>>
4. Compile
Component.c
Component.o

Figure 4: Common workflow in the automotive software development process [10]

change essential information he can easily do this in the source code where he probably has to change some other information.

When the information has been changed the new Software Component Description can be generated without changing anything else manually. For example, the modification of an interface would result in the modification of one line in the source code using the Software Component Generator. Applying system design tools this would result in the modification of the project file and the entire source code file.

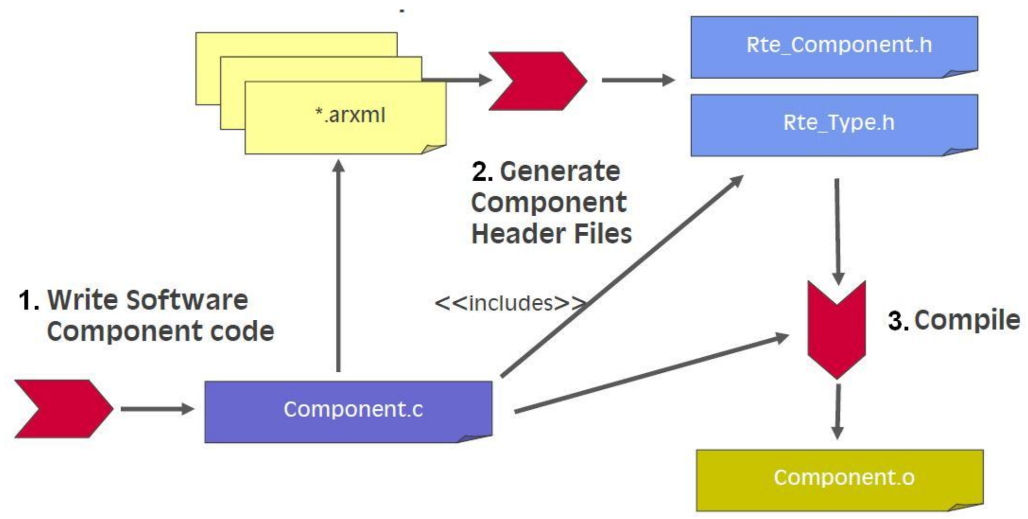

Figure 5: Requested workflow without writing Software Component Descriptions

3 Analysis of the Software Component Descriptions

This chapter deals with the syntactical and semantical analysis of Software Component Descriptions. A Software Component Description contains a lot of information about the Software Component [4]:

- Interface Descriptions: does the Software Component communicate with other Software Components, Basic Software modules or anything else? Which dataelements i.e. datatypes are transmitted? Which type of communication approach is being used?
- Atomic Software Component: which ports does the Software Component contain? Are these particular ports receiving or sending ports? Is it possible to instantiate more than one instance of this Software Component? (i.e. to use an measuring Software Component for the left and for the right door)
- Internal Behaviour: how are the particular Runnable Entities activated? Which Runnable Entity needs which port? Does the Runnable Entity access an Inter Runnable Variable or even an Exclusive Area?
- Compositions: which Software Components belong together? How do these Software Components interact with each other? Which elements will be transmitted between them?
- Implementation: where is the code that has to be executed? Is the code precompiled or not?

The following sections will analyse the particular elements that are called *AUTOSAR-Packages (AR-Packages)*.

3.1 Interfaces

Interfaces can be divided into two different types: Sender-Receiver-Interfaces and Client-Server-Interfaces. Both interface types have common attributes but differ in particular details as shown in Figure 6.

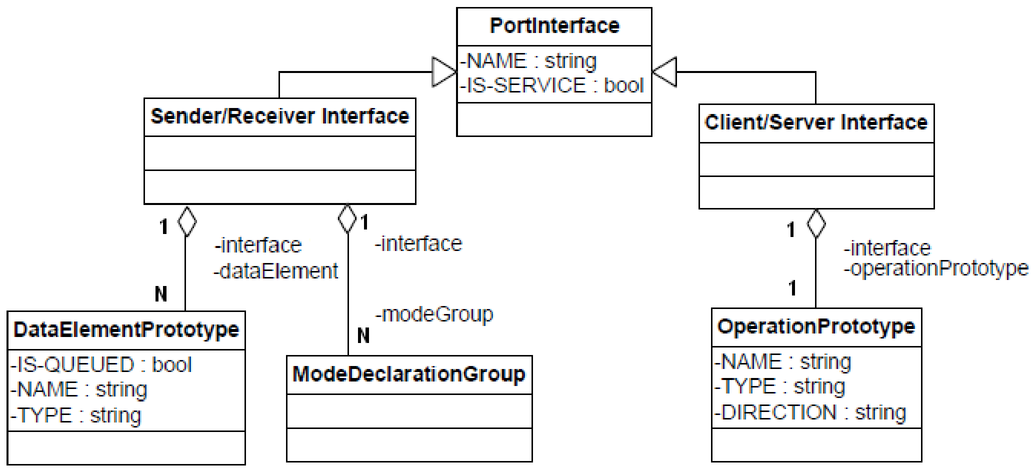

Figure 6: UML diagram of PortInterface [4]

A *Sender-Receiver-Interface* provides the most common way of communication. Data will be transmitted from one or more senders to one or more receivers. The communication is asynchron which means that the transmission takes only place in one direction. Transmitted packages will not be acknowledged.

A *Client-Server-Interface* is more likely to a *Remote Procedure Call (RPC)*. This implies a bidirectional communication approach in order to get the status from the called function. A Server provides one or more *Operations* with a fixed amount of arguments that have to be given. This *Server-Call-Point* can be either synchron or asynchron which ends either in a blocking or non-blocking server call.

Software Component Descriptions are given in XML. The formal structure of the Interface package can be written according to its UML specification in Figure 6. An example is given in Figure 7.

The example provides an Interface that aggregates one dataelement that is called `de1` from type `UInt8`. Additionally the dataelement will be transferred unqueued which means that the value will be overridden if the receiver does not process it fast enough (i.e. queue length = 1). API functions which needs to have access to Sender-Receiver-Interfaces are listed in Figure 8.

These functions are more than half of all API functions provided by the Runtime Environment and the common applied ones [6].

The functions shown in Figure 8 can be subdivided: functions containing the name of its Runnable Entity in it and the ones who do not. The last ones, are called *implicit* calls and do not change the dataelement during the Runnable Entity's execution. The Runnable Entity gets a copy of the sender's dataelement at the beginning of its execution and provides the result to the receivers not until its execution is over. The same takes place vice versa: the Runnable Entity provides the sent dataelement not before its

```
<AR-PACKAGE><SHORT-NAME>interfaces</SHORT-NAME><ELEMENTS>
    <SENDER-RECEIVER-INTERFACE>
      <SHORT-NAME>if1</SHORT-NAME>
      <IS-SERVICE>false</IS-SERVICE>
      <DATA-ELEMENTS>
        <DATA-ELEMENT-PROTOTYPE>
          <SHORT-NAME>de1</SHORT-NAME>
          <TYPE-TREF DEST="INTEGER-TYPE">UInt8</TYPE-TREF>
          <IS-QUEUED>false</IS-QUEUED>
        </DATA-ELEMENT-PROTOTYPE>
      </DATA-ELEMENTS>
    </SENDER-RECEIVER-INTERFACE></ELEMENTS>
</AR-PACKAGE>
```

Figure 7: XML of a PortInterface instance [3]

execution is over. Therefore the name of the related Runnable Entity has to be given in the function invokation.

```
Rte_Write_<port>_<dataelement> ( <type> data );
Rte_Send_<port>_<dataelement> ( <type> data );
Rte_Invalidate_<port>_<dataelement> ( <type> data );
Rte_Feedback_<port>_<dataelement> ();
Rte_Read_<port>_<dataelement> ( <type>* data );
Rte_Receive_<port>_<dataelement> ( <type>* data );

Rte_IRead_<runnablename>_<port>_<data> ( );
Rte_IWrite_<runnablename>_<port>_<data> ( );
Rte_IWriteRef_<runnablename>_<port>_<data> ( <type> data );
Rte_IInvalidate_<runnablename>_<port>_<data> ( );
Rte_IStatus_<runnablename>_<port>_<data> ( );
```

Figure 8: Function Defintion specified by the RTE [6]

Unfortunately, the function calls above are the only way to get information about the structure we need for the Software Component Description. The problem is, that the existence of underscores are the only way to separate the name of the port and the dataelement, which will probably lead to problems, because underlines may also occur within the name of the ports or the Runnable Entities.

With the fact that we can not determine the identifiers (e.g. name of the port, the dataelement and the Runnable Entity) even though it is regulated by the function definitions in Figure 8, we need to have additional information from the developer in order to complement the description:

- Name of the associated Interface. The name of the Interface has no relevance in the source code and will only be processed by the RTE.
- The name of the port **or** the dataelement. This is necessary because it is ambiguous at which point the underscore belongs to the portname or the name of the dataelement or functions as separator if there exist more than one underline.
- The datatype that is transmitted is not obvious for an Interface. It is determinable which datatype is given in the brackets, e.g. that will be returned by the function call, but it is not excludable that there will occur automatic typecasts. Hence it is not visible which type will be transmitted.

In case of implicit communication approach it is possible to separate the name of the Runnable Entity after parsing the entire sourcecode file.
Client-/Server communication is provided by fewer function possibilities:

```
Rte_Call_<port>_<operationprototype> ( arg1 , arg2 , ... );
Rte_Result_<port>_<operationprototype> ( arg1 , arg2 , ... );
```

As shown a Client-Server-Communication can not easily be devided into synchronous and asynchronous communication. If an *Rte_Result* will be invoked anywhere in the parsing area, it is obvious that the corresponding communication is asynchronous confirming to the specification. The decision whether the port is a sending- or a receiving-port can be determined as receiving. Providing Client-Server ports only occur in case of Server Runnable Entities which will be mentioned in 4.1. However, the arguments have to be specified in their direction. If there occur any variables that are dereferrenced, it can not be assumed that it will be an outgoing or incoming server-transmit operation, and it will be set to **INOUT**. A not derefferenced variable will be assumed as **IN**, because the communication will be triggered by-value. In case of Client-Server-Communication approach the needed metainformation is the same as mentioned before for the Sender-Receiver-Communication.

3.2 Components

As shown in Figure 9 the Components package provides information on undividable Software Components, the Atomic Software Components.

Atomic Software Components provide *RPortProtoypes (Require Port)* or *PPortPrototypes (Provide Port)*, which can be either part of a Client-Server-Interface or a Sender-Receiver-Interface. The Component package is in a way the *Black-Box-View* of the Software Component describing only the input and output connections.

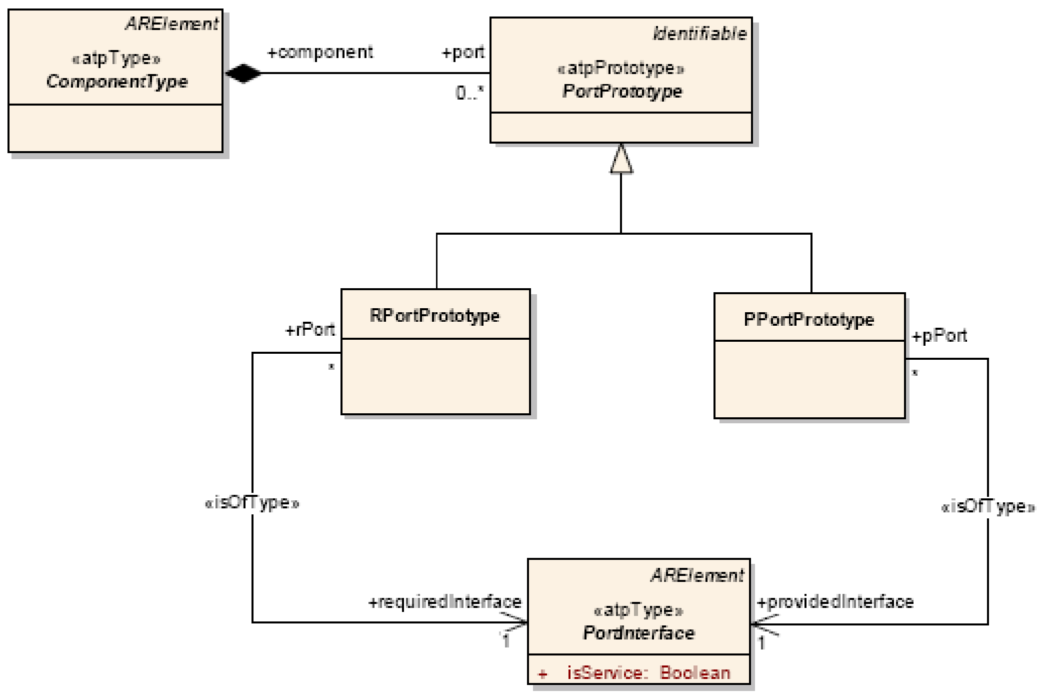

Figure 9: Software Component Template for the Component Prototype [4]

Summarizing, the only information that is needed for the Components package is the name of the Atomic Software Component and its used ports with its associated Interfaces. More complex information about each Atomic Software Component is described by the Internal Behaviour, which will be analyzed in the upcoming chapter.

3.3 Internal Behaviour

The Internal Behaviour is the most complex part of the Software Component Description. Each Atomic Software Component is adviced to have only one Internal Behaviour. According to Figure 10 the Internal Behaviour consists of the following main elements: Runnable Entities, RTEEvents, Exclusive Areas, RunnableExecutionConstraints, PerInstanceMemory (PIM). The Internal Behaviour consists of even more elements that are not part of the UML diagram. However these elements are not of interest and would only distort the diagram. The following subchapters will introduce the main members of the Internal Behaviour in the same order as they have to appear in the Software Component Description.

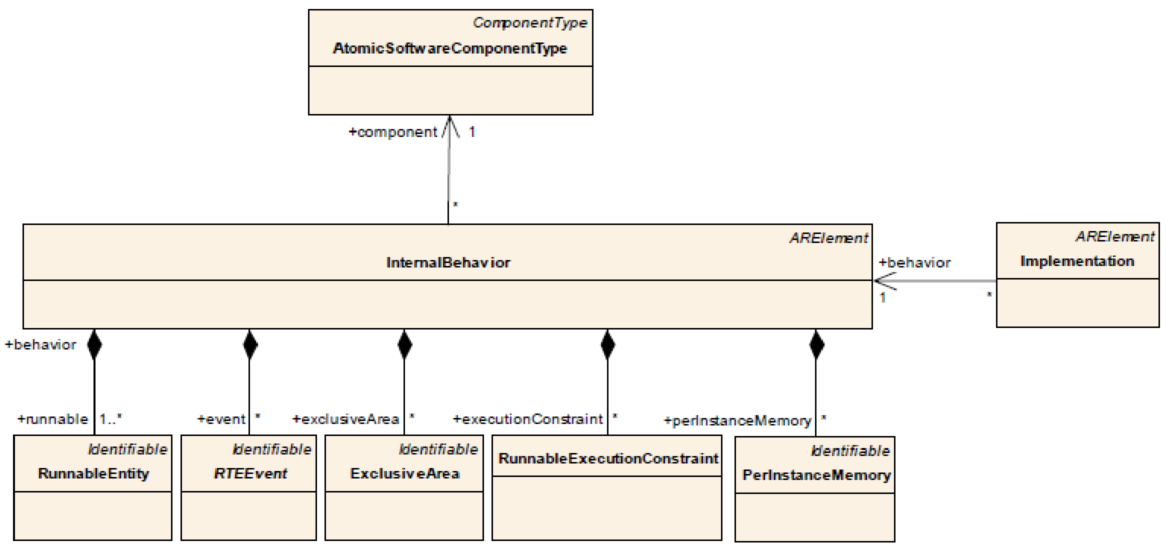

Figure 10: Meta-model of the Internal Behaviour [4]

3.3.1 Events

Events trigger Runnable Entities. They are differently defined as own elements in the main section of the Internal Behaviour. These elements will then be referenced within the XML definition of the Runnable Entity. These events can be from different types:

- **AsynchronousServerCallReturnsEvent:**
  "The event is raised when an asynchronous server call is finished" ([4] p. 106).
- **DataSendCompletedEvent:**
  "This event is raised when the referenced data element have been sent or an error occurs" ([4] p.100).
- **DataReceivedEvent:**
  "This event is raised when the referenced data elements are received" ([4] p. 101).
- **DataReceiveErrorEvent:**
  "This event is raised by the RTE when the communication layer detects and notifies an error concerning the reception of the referenced data element" ([4] p. 103).
- **OperationInvokedEvent:**
  "The OperationInvokedEvent references the operation invoked by the client" ([4] p. 107).
- **TimingEvent:**
  "A TimingEvent references the Runnable Entity that needs to be started in response to the TimingEvent" ([4] p. 108).

ModeSwitchEvent

ModeTypes can be used in order to transmit status information. For instance, the ModeTypes *ignition on* and *ignition off* can be transmitted to several Interfaces such that ECUs can then start or disable services. Consequently whenever a Mode is entered or exited a Runnable Entity must be started ([4] p. 24)

A TimingEvent is the most common applied TriggerEvent. A majority of jobs that have to done on an ECU will require periodic actions. Figure 11 focuses on the proper XML code of the Timing Event.

```
<TIMING-EVENT>
  <SHORT-NAME>TimingEvent_10ms</SHORT-NAME>
  <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">/internals/IB_A/
                   RunnableA</START-ON-EVENT-REF>
  <PERIOD>0.01</PERIOD>
</TIMING-EVENT>
```

Figure 11: XML declaration of a TimingEvent [3]

Another huge part of Events are the ones with data semantics, which have to do with communication mechanisms. These events are all similar defined and only differ in the declaration tag of the Event. As shown in Figure 12 the event is simply described by the port and the dataelement.

```
<DATA-RECEIVED-EVENT>
  <SHORT-NAME>drp1</SHORT-NAME>
  <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">/internal/IB_ASC/
                           RunnableA</START-ON-EVENT-REF>
  <DATA-IREF>
    <R-PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">/comp/ASC/
                           port</R-PORT-PROTOTYPE-REF>
    <DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-PROTOTYPE">
                 /inter/if1/de1</DATA-ELEMENT-PROTOTYPE-REF>
  </DATA-IREF>
</DATA-RECEIVED-EVENT>
```

Figure 12: XML declaration of a DataReceivedEvent [3]

The OperationInvokedEvent is nearly the same as the DataReceivedEvent. It is derived from a Client-Server-Interface instead of a Sender-Receiver-Interface. In addition to that the associated Runnable Entity gets the arguments from the referenced *Operation Prototype* as parameters. Such a Runnable Entity is called *Server Runnable Entity* and is more precisely described in 4.1.

3.3.2 Inter Runnable Variables

The RTE provides a simple Sender-Receiver like communication between Runnable Entities that are associated with the same Software Component. The access to this variable is certainly much faster than a normal communication approach. One can indicate the usage of Inter Runnable Variables through the following functions:

```
Rte_IrvWrite_<re>_<name> ( value );
value = Rte_IrvRead_<re>_<name> ( );
Rte_IrvIWrite_<re>_<name> ( value );
value = Rte_IrvIRead_<re>_<name> ( );
```

The occurence of one of these functions leads to the XML correspondency which is shown in Figure 13.

```
<INTER-RUNNABLE-VARIABLE>
  <INTER-RUNNABLE-VARIABLE>
    <SHORT-NAME>name</SHORT-NAME>
    <TYPE-TREF DEST="INTEGER-TYPE">/PATH/TO/DATATYPE</TYPE-TREF>
    <COMMUNICATION-APPROACH>IMPLICIT</COMMUNICATION-APPROACH>
    <INIT-VALUE-REF DEST="INTEGER-VALUE">/PATH/TO/VALUE</INIT-VALUE-REF>
  </INTER-RUNNABLE-VARIABLE>
</INTER-RUNNABLE-VARIABLES>
```

Figure 13: XML of Inter-Runnable-Variables [3]

In addition to that declaration, which has to be set right in front of the Runnable Entity's section, the usage of this Inter Runnable Variable has to be shorty notified within the Runnable Entity's by typing the lines shown in Figure 14.

```
<READ-VARIABLE-REFS>
  <READ-VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">
    /internals/IB_ASC/irv
  </READ-VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">
</READ-VARIABLE-REFS>
```

Figure 14: XML of referenced Inter-Runnable-Variable [3]

3.3.3 Exclusive Areas

The RTE provides a mechanism to synchronize the access to Exclusive Areas. Exclusive areas are critical sections and lead to preemption-avoidance if a Runnable Entity enters or exits it. An Exclusive Area ensures the atomical access to a resource. This can be indicated by using the following two API functions:

```
void Rte_Enter_<name> ();
void Rte_Exit_<name> ();
```

These functions synchronize the access among Runnable Entities that are related to the same Atomic Software Component. This simply is indicated by the following XML-code [3]:

```
<EXCLUSIVE-AREA>
  <SHORT-NAME>name</SHORTNAME>
</EXCLUSIVE-AREA>
```

Now, all Runnable Entities within this Internal Behaviour have the ability to use this Exclusive Area. This privilege will be granted by the following piece of code that has to be placed within a Runnable Entity [3]:

```
<RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREA>
  <EXCLUSIVE-AREA-REF DEST="EXCLUSIVE-AREA">/internalbehaviours/
                IB_Atomi/area</EXCLUSIVE-AREA-REF>
</RUNNABLE-ENTITY-CAN-ENTER-EXCLUSIVE-AREA>
```

The mechanism of how to add the ability to use Exclusive Areas is exactly the same as for the Inter Runnable Variables: defining the Exclusive Area in the main section of the Internal Behaviour and referencing it within the Runnable Entity.

3.3.4 Runnable Entities

Each Atomic Software Component consists of at least one Runnable Entity that executes code.

A Runnable Entity serves to structure the Software Component's functionality. Each Runnable Entity can be scheduled or triggered individually and independently from other Runnable Entities. Each Runnable Entity should be triggered by at least one Event. Runnable Entities are implemented by C-functions. The name of this function is the *symbol*-attribute of the Runnable Entity.

The Software Component Description defines the Runnable Entity with a huge amount of elements (which also can be seen in Figure 15):

- **Name:** the name of the Runnable Entity (not the C-function name!)

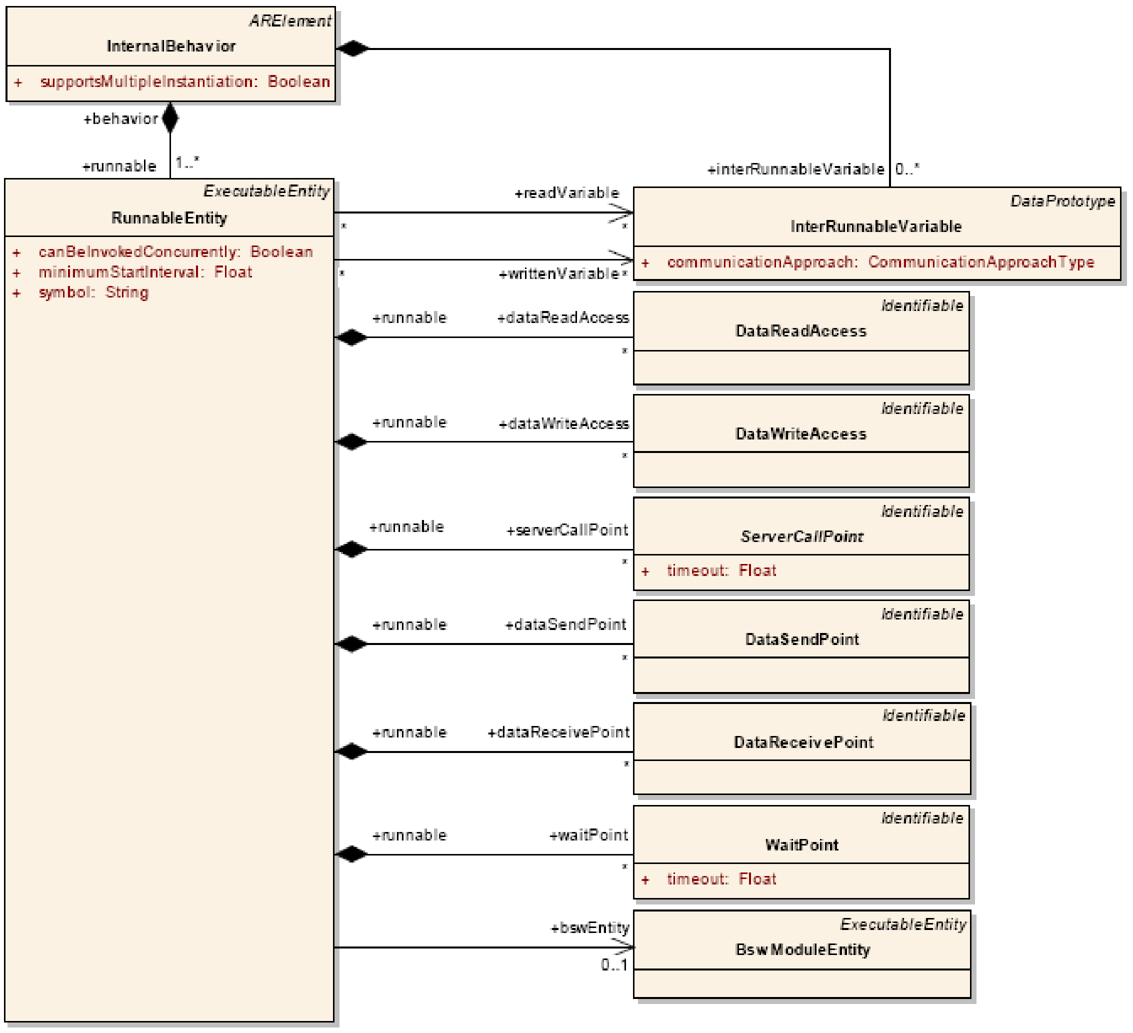

Figure 15: UML diagram of the Runnable Entity [4]

- **Can-Be-Invoked-Concurrently:** true, if the Runnable Entity can run more than once at a time, false otherwise. This could be useful for example if a Server Runnable Entity can be invoked by more than one client at the same time.

- **Server-Call-Points:** a list of server-call-points, which is needed if the Runnable Entity invokes a *Rte_Call*.

- **DataTransmitPoints:** data transmit points can either be DataSendPoints or DataReceivePoint, which are necessary in case of using the functions *Rte_Write*, *Rte_Send*, *Rte_Receive* or *Rte_Read*. Data Transmit Points reference a port and the port's associated Interface.

- **Data-Write/Read-Access**: this "points"are needed in case of implicit communication modes. Generally the definition of this points is the same as the normal DataTransmitPoints.

- **Uses-Exclusive-Areas:** the Runnable Entity has the ability to enter an Exclusive Area. Therefore the Runtime Environment has to ensure synchronized access.

- **Written/Read-Variables:** when the Runnable Entity accesses Inter Runnable Variables, which has been defined above, it will be referenced here.

If a C-function is determinable as a Runnable Entity, it is possible to get all Data Transmit Points, Exclusive Area usages and Inter Runnable Variable usages by tracing the control flow of the function. However there is some information that can not be determined by simply looking at the code itself:

- A C-function has to be marked as a Runnable Entity, otherwise it would not be possible to determine the other information.

- Each Runnable Entity has to be associated with an Atomic Software Component. This can be recognized for the very reason that the headerfile of an Atomic Software Component has to be included anywhere a Runnable Entity is defined.

- Trigger event: Tells the RTE, how the Runnable Entity has to be triggered. This can either be a perodic time event (for example 10 ms) or a data event (if data is received, the Runnable Entity should be invoked to process this data).

- Concurrent invokation: defined when the Runnable Entity can run more than once at the same time, i.e. runs concurrently.

With this information it is possible to set up the complete Internal Behaviour by tracing the control flow.

3.4 Compositions

The Composition package connects Atomic Software Components and its Internal Behaviours and connects ports from a Software Component, if necessary. At least one *Top Level Composition* is required to instantiate Software Components. Top Level Compositions will not be aggregated to further Compositions any more. It will not be possible to run Atomic Software Components if they are not associated to a Composition. The Composition also provides important information about the connection of ports within it. To connect two ports, e.g. from one Atomic Software Component to another within the same composition, an *Assembly Connector* can connect a Provide-Port with

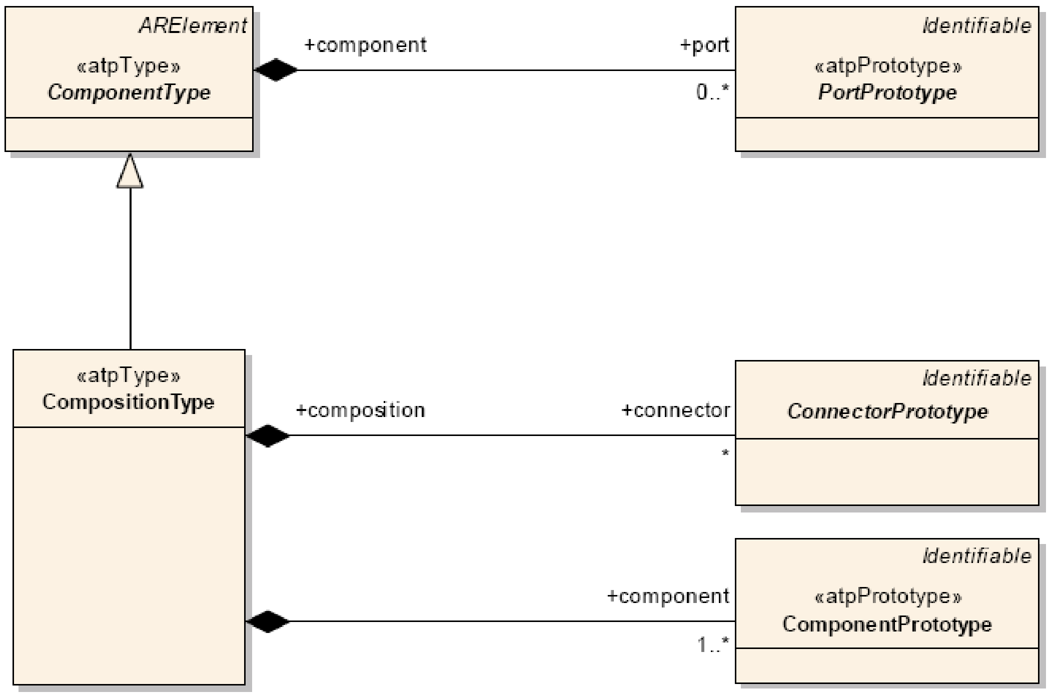

Figure 16: UML diagram of the Composition Package [4]

a Require-Port. If a port should be made available from or to the outside of its Composition, a *Delegation Connector* is needed to connect the port. The UML reference of the Composition package with its connectors and components can be seen in Figure 16.

Having to define these Connectors at design time leads to a crucial conclusion: all communication partners have to be known at generation time. Contrary, during the development of the Software Component, it is not necessary to know which component runs on which ECU - the RTE passes data to the right Software Component even if this means to pass the data out of the current ECU.

Moreover the Composition provides information about the implementation of each Atomic Software Component or Internal Behaviour. That contains information about the programming language, path of the Internal Behaviour within the Software Component Description and resource consumption.

Despite the fact that the amount of information within the Composition package is low, it will not be implemented within this thesis. Hence we have no information about the Composition in the code, we need all this information from the software developer. For this reason it will be better if the software developer writes compositions for its Software Components himself.

3.5 Summary of Analysis

The appearance of API function tells much about Runnable Entities, name of ports and dataelements and - depending on the function - further information (like name of Inter Runnable Variables and so on).

Conclusively the goal for the parser and the generator is to read the information and to create a semantically correct context (at first the connection to a Runnable Entity and then to an Atomic Software Component). Missing information can be retrieved of metainformation which is absolutely affordable in consideration of not having to design a Software Component Description any longer.

Hence it it not determinable which Interface a port belongs to or which (complex) datatype is used, it is possible to read out a lot of information and translate it to a valid, well-formed and semantically correct Software Component Description.

4 Design and Implementation

The design and implementation were the most tremendous part of this thesis. The tools should be designed in a way that adding new API functionality and parser information is an easy task. Therefore, it is clear that the interface of the generated parser and the generator have to be defined clearly. Afterwards, adding new functionality means adding code in the parser and in the code generator.

4.1 General Syntax Structure of the Meta Language

As we have seen in the previous chapter, some metadata is needed to generate a complete Software Component Description. Summarizing, there are exactly four cases in which additional information is necessary:

1. At the function declaration point of a Runnable Entity.
2. At each point where communication appears (i.e. *Rte_Read, Rte_Write...*).
3. At points of mode switches of ports (ModeTypes will be describes within the upcoming subchapters).
4. In case of the incidence of unknown datatypes, which have been pre-compiled for example, it is necessary to give information about these datatypes.

In the following part the metadata, which is used for the 4 points is shortly introduced.

Runnable Entity

```
//@ runnable <NAME>, <TRIGGER-EVENT-TYPE>, <CONCURRENTLY-INVOKABLE?>
void RunnableA(<Rte_Instance> self?) { ... }
```

NAME	The name (not the symbol!) of the Runnable Entity.
TRIGGER-EVENT-TYPE	Either triggered by DataReceivedEvent() or triggered by TimingEvent().
CONCURRENTLY-INVOKABLE?	Can be either *true* or *false.*
Rte_Instance	Omitted if the Atomic Software Component is not multiple instantiatable.

Server Runnable Entity

Server Runnable Entities are Runnable Entities that are invoked by an *OperationInvokedEvent*. This Event is characterized by a Port Prototype and an Operation Prototype. This Runnable Entity then gets the arguments of the associated Operation Prototype as parameters. Hence, Runnable Entities that are invoked by such an Event have to be parsed slightly different. In fact, a Server Runnable Entity is a function that can be used for Remote Procedure Calls.

Therefore the metainformation is the same as it is for other Runnable Entities, just the Event has to be switched from an *DataReceivedEvent* to an *OperationInvokedEvent*. Having parsed the OperationInvokedEvent, the parser has to determine which arguments the Runnable Entity function gets in order to build the Operation Prototype arguments.

In addition to that, the Interface associated with this event can simply be a pseudo interface. This can easily be generated because the Interface will not and can not be applied for anything else within this SW-C. The port associated to that Interface will be a Provide Port. With the fact that it is not possible to create a second Runnable Entity that is triggered by the same OperationInvokedEvent this Operation will only be offered once. Afterwards, it is possible to create Server Runnable Entities the same way as other Runnable Entities have been created. The following shows an example implementation:

```
//@ runnable Server, triggered by OperationInvokenEvent(de, OP), false
void max(Rte_Instance instance, int a, int b, bool &c) { ... }
```

S/R and C/S-Communication

```
//@ interface <IF-NAME>, datatype <DATATYPE>, <DATAELEMENT> [,<SERVICE>]
Rte_Write_<PORT>_<DATAELEMENT>( Rte_Instance <self>, ... );
```

IF-NAME	The name of the related Interface
DATATYPE	The datatype of the dataelement that has to be sent/received (i.e. UInt16 etc.).
DATAELEMENT	The name of the dataelements as it occurs in the function header itself (This has to be given because it is not obvious to separate the portname and the dataelementname in the function call, because in case of more than one underscore it is not ensured to whom the underscore shall belong. See Figure 17 for details.).
SERVICE	If the related Interface is a Service Interface (i.e. a provided Interface given by AUTOSAR Basic Software), this value has to be set *true* else it is not necessary and set on false by default.
PORT	The name of the port.

```
Rte_Write_Port_Data_DE1 ( 1 );
```

Figure 17: Multiple possibilities for Port and Dataelement

Figure 17 illustrates the problem for the parser to determine the both names automatically. According to the example there are two options that are possible:

1. Portname `Port`, Dataelement: `Data_DE1`
2. Portname `Port_A`, Dataelement: `DE1`

Hence it is not possible to commit the task of setting the names to the software. The developer has at least to specify one of them.

Server Call Points with different Operation Prototype

An Operation Prototype that is defined for a Client-Server-Interface includes a variety of dataelements which have to be given as arguments exactly in the same sequence as they are defined.
Such Server-Call-Points could be indicated as follows:

```
Rte_Call_Port_Operation1(instance, value, &value, true);
Rte_Call_Port_Operation2(instance, &temp, speed);
```

In this case we have more problems at a time: *instance* can be an argument for the OperationPrototype *Operation1* or *Operation2* but only if the Atomic Software Component does not support multiple instantiation. If the Atomic Software Component supports multiple instantiation, the instance handle, which is given to the Runnable Entity, has to be passed through all API functions as the first parameter and is not part of the Operation.

In addition to that, the values given to the function can be from type *IN, OUT* or *INOUT*. For instance, a variable that is not dereferenced in the argument list, will be an incoming parameter for the Server Interface, because the parameter will be passed by value. If the application developer wants to get a result from a Server Runnable Entity, he or she has to pass a variable with prefixed "&". In this case, the argument can be both an incoming argument or an *INOUT*-argument. It can never be ensured that it only will be an outgoing argument.
Whereas it is possible to pass a greater amount of parameters, the metainformation has to be extended as follows:

```
//@ interface if1, datatype ( UInt32, UInt32, Boolean ), Operation1
Rte_Call_Port_Operation1(instance, value, &value, true);

//@ interface if1, datatype ( SInt8, UInt8 ), Operation2
Rte_Call_Port_Operation2(instance, &temp, speed);
```

Now it is possible to extract the information of all Operation Prototypes.

ModeTypes and ModeSwitch

Mode switches indicate an Interface that a mode switched from one ModeType to another. For instance, that can occur for the states "ignition on"and "ignition off". This simplifies the work for the application development in a couple of cases. An example can be:

```
//@ Mode PortA Ignition
Rte_ModeType_IgnitionOn mode1;
//@ Mode PortA Ignition
Rte_ModeType_IgnitionOff mode2;
...
Rte_Switch_PortA_Ignition(mode1);  // Switch to IgnitionOn-Mode
```

PORTNAME	The name of the Port that has to be assigned to this Mode.
MODE_PROTO	The name of the ModePrototype.
M	The name of the ModeType.

When applying ModeTypes it is possible to trigger Runnable Entities on Mode-Switch-Events. Whenever any ECU switches a Mode (e.g. *ignition on* to *ignition of*) the associated Interface will be noticed and a Runnable Entity can be triggered to process data (in this case to shut down the ECU properly without losing volatile data).

Unknown Datatypes

```
//@ datatype <TYPE>
Foo bar;

bar = Irv_Read....
```

The *TYPE* can be either a C or an AUTOSAR standard datatype, which is *int, char, UInt16, UInt32, ...* The generator follows this definitions later on and will write the last one into the XML file. In case of incomplete definition, this will be an unknown datatype and will fail the generation process if it can not be resolved.

4.2 Meta Language And Parser Implementation

The aim of the parser is not to check C syntax but to extract metainformation. It is not possible to ommit the complete syntax as described below, but most of the syntax information is not necessary. The main grammatical structures that are interesting for the parser are: metainformation, declarations (functions and variables), RTE API functions, function call points and function declarations.

In each of the cases above detailed information is necessary to be parsed. If a phrase or statement is not of any of the types above, it will be parsed properly but not considered any further.

4.2.1 Core points and key features

Figure 18 shows the workflow of the code generator. The following subchapters will present problems and solutions that were made during the pre-processing and the parsing steps.

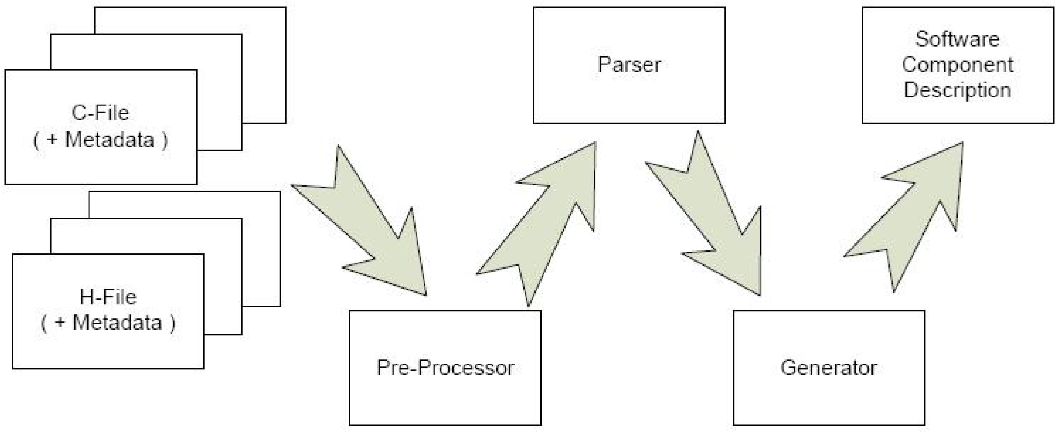

Figure 18: Flow within the generator software

Pre-processor

In order to address the problem that functions and variable declarions can be written in other files, it is necessary to do a step before starting the parser.
It is necessary to open the main file and displace *#include* statements with the content of the actual file(s) recursively but only if the file has not been included yet. This is efficiently done by only a couple lines of code showed in Figure 19.

Afterwards the pre-processor created a temporary file that contains the content of the original file and all included header files. If the header file does not exist or is included from the default include folder, it will not be processed. By now all the information that has to be parsed is placed within one single file.

Variable Scopes

To determine the return type of various functions (like Inter Runnable Variables) it could be useful to automatically detect the datatype in order to get reduct metainformation. In other words: the datatype of an Inter Runnable Variable will be detected automatically.

```
  // the main program file has to be opened before
  public void outputFile(String filename, PrintStream out,
          BufferedReader in) throws Exception {
    while (in.ready()) {
      if (line.startsWith("#include\"") {
        // extract filename
        if (!Included(line)) { // check if already included
          // Create BufferedReader on filename
          outputFile(line, out, bufferedreader);
        }
      } else {
        out.println(line);
  } } }
```

Figure 19: The Pre-Processor implementation

The problem encountered here is to handle variable scopes. A variable consists of two strings: datatype and typename. The parser now parses the C-file and stores all variables that has been parsed in an *ArrayList*. The problem is that variables will temporarily lose their validity if they are overridden in another (deeper) scopes.

```
String tmp;

void foo() {
  int tmp = 2;
  if (true) {
   char tmp = 'a';  // tmp is from type 'char'
  }  // tmp is from type 'int'
}  // tmp is from type 'String'
```

Figure 20: Validity scopes of temporary variables

To face the problem focused in Figure 20 it is necessary to count the depth of scope (within nested curly brackets) and save this value, too. Then it is possible to exactly determine which datatype is returned by a couple of functions.

Function Control Flows

The RTE has to know all the resources that will be used during runtime by each Software Component. In order to provide synchronization within a Software Component and for further purposes, the RTE also has to know which resources will be used by every single Runnable Entity. This leads to the fact that the parser also needs to know which Runnable Entity and which function it is about to parse as well as the methods this function calls. This is inevitable in order to get a full list of API calls for each particular Runnable Entity.

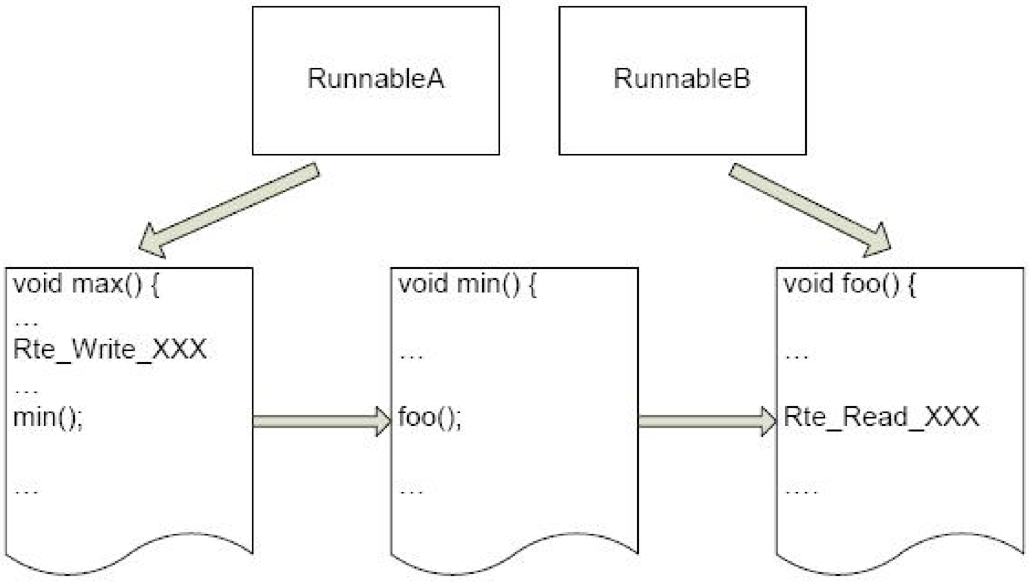

Figure 21: Illustration of port and interface usage

In the case illustrated in Figure 21, *RunnableA* needs the ability to access the port *RunnableB* has access to. To address this problem, the parser has to remember the current function it is about to parse and the available functions, which has been declared somewhere else in the code.

Each *RteTupel* (a structure holding all information about API function calls) has a member variable *daddy_function* that provides information about the function within it occurs.

This is done for each API function in order to create an invokation tree (which fulfills not necessarily the definitions of a tree) at the end of the parsing step by implementing the following lines:

The `buildtree()` function shown in Figure 22 processes all functions that are marked as Runnable Entities and sets them as entry points. The `getChild()` function steps through all called functions recursively and avoids loops if functions are already tra-

```
public void buildTree() {
  for(Function func: functions) {
    if (!func.isRunnableEntity()) getChild(func, func);
  }
}

public void getChild(Function a, Function runn) {
  if ((a.getFktChilds().size() == 0) &&
            (a.getRteChilds().size() == 0)) return;
  for (Function re: a.getFktChilds()) {
    getChild(re, runn);   // avoiding loops here
  }
  for (RteTupel re: a.getRteChilds()) { // assigning Tupel
    global_swc.assignTupel(re, runn.getName());
  }
}
```

Figure 22: Call tree construction

versed. The result is a completely traversed call tree. The tree for the given example is shown in Figure 23.

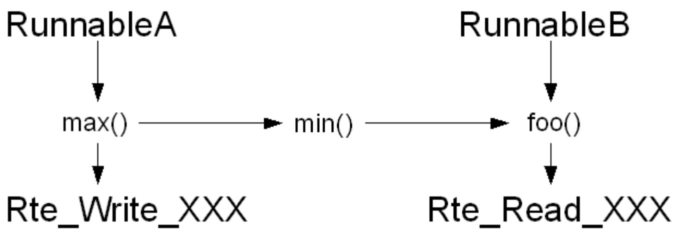

Figure 23: Created call tree after generation process

Pre-processor instructions

Pre-processor statements occur simplified in two slightly different ways:

- Defining a function macro:

```
#define FUNCTION(A, B) A+B
```

- Defining constants

```
    #define PI 3.14159
```

The definition of constants is nearly the same as the definition of normal variables. Although due to the fact that it is not possible to process preprocessor macros it is not possible to resolve these *#defines*. The following example shows a problem when dealing with preprocessor instructions:

```
#ifdef DEBUG
#define SOMETHING 0
#elsif
#define SOMETHING 1
#endif
```

Therefore, the parser does not implement such macros and will ignore them.

AUTOSAR defined datatypes macros

The AUTOSAR compiler abstraction defines a couple of macros to determine the actual datatype respective to the ECU configuration.

There will be provided a couple of function macros, each for particular circumstances. Unfortunately, it is not possible to resolve these macros, because they are not defined in any given headerfile. They will be provided later on (after the generation process) by the RTE.

```
// used for return types
FUNC(uint8, RTE_APPL_CODE) functionname

// used within parameter list of function definition
P2CONST (TYPE, AUTOMATIC, TYPE_APPL_CONST) variablename

// used to define variables
VAR (TYPE, AUTOMATIC) variablename;
```

Figure 24: AUTOSAR defined datatype macros [5]

However, the actual datatype that will be defined is given as the first argument of the parameter list of each macro as shown in Figure 24 . Hence, it is possible to simply take the first statement and ignore the rest. This works properly in the present case.

4.3 SWC Generator Implementation

After having done the pre-processing and the actual parsing, it is now possible to process the data. The main program just does the following couple of steps as shown in 25.

```
SwcManager global_swc = new SwcManager();

try {
  PreProcessor pre = new PreProcessor(cfile);

  CLexer lex = new CLexer(new ANTLRFileStream(cfile + ".tmp");
  COmmonTokenStream tokens = new CommonTokenStream(lex);
  CParser g = new CParser(tokens);

  g.initialize(global_swc);
  g.translation_unit();
} catch (...) { ... }
```

Figure 25: The application of the SwcManager in the main program

The parser gets an object of the type *SwcManager* and fills in all the data it parses. Therefore, the SwcManager (e.i. the generator) provides a variety of functions to retrieve the data needed to be converted to a Software Component Description. This functionality will be explained in the following subtopics.

4.3.1 SWC Generator Class Overview

The SwcManager interact with a variety of classes that can be seen in Figure 26. These classes mainly function as data containers in order to hold the structure of the current parsed information as clear as possible. The functions provided by the SwcManager can be invoked for any information that is parsed. They will be explained in the following.

- `void addExlusiveArea(String name, Function current)`

  This function is being used to add the incidence of an Exclusive Area under the contect of the *current* function.

- `void addDatatype (String from, String to)`

  In case of variable definitions, typedefs or defines this function sets up a list of mappings for each occuring variable.

- `void addAsyncResult (String str)`

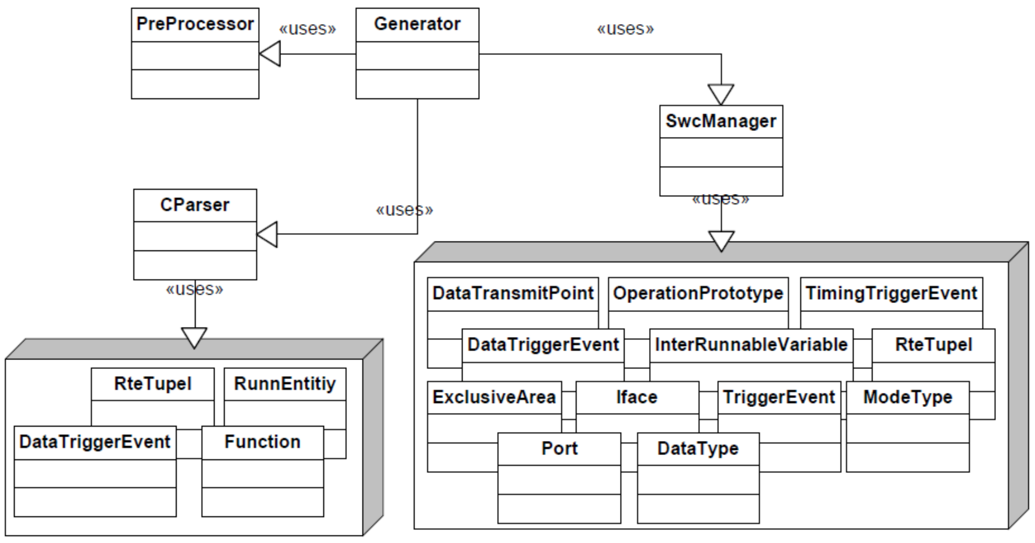

Figure 26: Overview of the used classes

If an *Rte_Result* occurs while parsing, it means that the corresponding *Rte_Call* is asynchronous. This value has to be cached until the end of the parsing because the associated *Rte_Call* has not necessarily been parsed yet. If one wants to have an asynchronous communication mode and there is no *Rte_Result* occurence in the codefile (which can happen in case of precompiled code parts), this statement has to be somewhere in the code even though it is not called.

- `String findDataType (String from)`

  This function takes a variable name and searches the internal list of datatype for the origin datatype. This can be more than one mapping regarding the incidence of typedefs.

- `void addIRV (String spec, String rest, Function daddy, String datatype)`

  If an *InterRunnableVariable* was found by parsing `Rte_Irv[...]`, this will simply be noted by this function. The structure of the variable will be set later on (except of its datatype) right before the export.

- `void addCommunication (String spec, String if_name, Boolean isService, String datatype, String rest, Function daddy_function)`

  This function is always called whenever any kind of communication occurs in the code. It is the task to adjust the right attributes by distinguishing between Sender-Receiver and Client-Server-Communication or other ones. In addition to that, it checks whether the function has incoming parameters (which is indicated through

"&" in front of variable names in the parameter list) and extracts the name of the port from the name of the Dataelement Prototype or Operation Prototype.

- `void addRunnEntity (RunnEntity re)`

  If the parser parses the beginning of a Runnable Entity it will instantiate an object of type *RunnEntity* and the SwcManager will simply add this object to its list.

- `void addMultipleInstantiation (String atomic)`

  If a Runnable Entity has a parameter from type *Rte_Instance*, this Atomic Software Component will be multiple instantiatible.

- `Iface addInterface (String name, InterfaceType t)`

  The behaviour of this function is similar to the behaviour of adding a Runnable Entity. The function is called whenever metainfo and an associated API call occured. If the interface is already parsed it will be ignored. The *InterfaceType* can either be *SwcManager.CS* or *SwcManager.SR*.

- `void assignTupel(RteTupel rte, String re)`

  At the end of the parsing each RTE function has to be assigned to one or more (or in some cases even none) Runnable Entity by tracing the control flow. This function finally sets a reference to the *RteTupel* is the associated Runnable Entity `re`.

- `void addModeType (ModeType m)`

  Modetypes can not be defined more than once. Therefore it is only necessary to add the existence of this modetype to a list and proceed this list later on.

- `void export (PrintStream out)`

  This function applies a variety of data in several dataelements, which is needed to create the Internal Behaviour.

  1. Creating asynchronous *ServerCallPoints*. This is done by searching the corresponding *Rte_Call* to each parsed *Rte_Result*.
  2. Creating of communication points. Each Runnable Entity has to know which *DataTransmitPoints* occur in their current control flow.
  3. Building internal structures of Inter Runnable Variables, Data Access Points and Exclusive Areas

  After building this structures above, the function calls private members that export the particular *AUTOSAR-Packages*. These functions are:

```
    private void exportComponents(PrintStream out);
    private void exportInternalBehaviour(PrintStream out);
    private void exportInterfaces(PrintStream out);
    private void exportImplementation(PrintStream out);
```

4.3.2 Generation of XML-File

The export of the resulting Software Component Description as XML is the final step that has to be taken by the generator. This also includes the evolvement of missing data which has to be fetched out of other information first. The next step is to generate the particular *AR-PACKAGES* subseqeuently, one after another.

In general, there are six important packages that need to exist to get the software running on an ECU:

1. Interfaces
2. Components
3. Internal Behaviours
4. Implementation
5. Datatypes
6. Compositions

The first four packages can be extracted as mentioned before. The Composition package requires too much metainformation not to say only metainformation and has no relation to the code itself. Hence, it is not possible to automatically create this package. Notwithstanding, four packages can be derived. This will be explained in the following.

Generation of Interfaces

Having a list of concrete structured Interfaces, it is easy to write the necessary information out as XML. Going through the list of available Interfaces it is required to determine whether a particular Interface is from Sender-Receiver (SR) type or Client-Server (CS) type. For both the CS and the SR Interface the general information is the same. In case of Sender-Receiver-Interfaces it is necessary to export all dataelements and mode types that are aggregated. The Client-Server-Interfaces do not own dataelements or mode types but Operation Prototypes. Each Operation Prototype consists of an amount of arguments that will be transmitted by calling this operation. Both Interface types can be handled similar.

Generation of Components

The Atomic Software Components have less XML code than the Interfaces but afford more computational work. For each Atomic Software Component a list of needed ports has to be created. This is realized by looking at each available Runnable Entity and taking these ports.

Then it is possible to create one port description after another by sequentely stepping through the port list. It does not matter whether the port is a PPort or a RPort or even belongs to a Client-Server- or Sender-Receiver-Interface. The syntax of these different types looks very simlar and does not result in great differences in XML. The only matter that has to be distinguished is if the dataelements appear queued or unqueued. This depends on which API function is being used (For example `Rte_Write` is unqueued and `Rte_Send` is queued). This results in a queued and an unqueued list within each the providing/receiving-communication-specification tags within the port descriptions.

If constants are used (like for the *INIT-VALUE* of dataelements), they could be defined in the Components section right after the defintion of the Atomic Software Components.

Generation of Internal Behaviour

The generation of the Internal Behaviour requires effort due to the fact that parts of it are referenced within the entire Software Component Description.

The Internal Behaviour consinsts of triggering Events, Exclusive Areas, Inter Runnable Variables and Runnable Entities, which also contain data receive/send points, data read-/write accesses and server-call points. All these information will be generated within the export function for the Internal Behaviour (the structure of the used objects is generated before in the global export function):

```
for (RunnEntity re: runnables) {
  /* fetch Trigger Events that are used for Runnable Entities
     within this Atomic Software Component  */
}
```

The list resulting from that piece of code allows to write out all trigger events easily, e.g. *TimingEvent, OperationInvokedEvent, DataReceivedEvent*. At this point the generator also checks whether this event is suitable (a DataReceivedEvent would be useless if the corresponding interface does not exist) or not. If a dataelement which is connected to the event is not defined within an interface that is part of the control flow of a Runnable Entity there will be a warning. The generation process for the Exclusive Areas and the Inter Runnable Variables processes the same pattern.

Afterwards, each Runnable Entity will be exported under the same Internal Behaviour due to the fact that it is recommended to only apply one Internal Behaviour. Each Runnable Entity contains a list with all of its data transmit points, which are already distinguished in Data-Send-Points, Data-Receive-Points, Data-Read-Accesses and Data-Write-Accesses, which indicate that the Runnable Entity provides/requests a new value. The value will not be modified during execution. These Data Transmit Points will be created in the previous export step. All these elements will be subsquently written out step by step for each Runnable Entity.

Finally, it is time to export the used Inter Runnable Variable distinguished in read access and write access to this variable:

```
for (InterRunnableVariable irv: this.irvs) {
  if ((irv.getRunnableEntity().getName().compareTo(
                                    re.getName()) != 0)
         && ( irv.isReading()) )
     out.println("<READ-VARIABLE-REF DEST=..........");
  }
}
```

The same piece of code will also be processed for the written variable accesses. After that, the server-call-points (divided in asynchronous and synchronous) and information about the usage of Exclusive Areas will be exported.

5 Evalation

In this evaluation a simple regulatory circuit is implemented. Imagine the following scenario: an integrated circuit produces a considerable amount of heat that has to cooled down if it exceeds a maximum of 50 degrees. Otherwise the circuit will work inefficiently if its working temperature will fall down below a minimum of 40 degrees. This is a typically occuring problem that can be addressed using cybernetics.

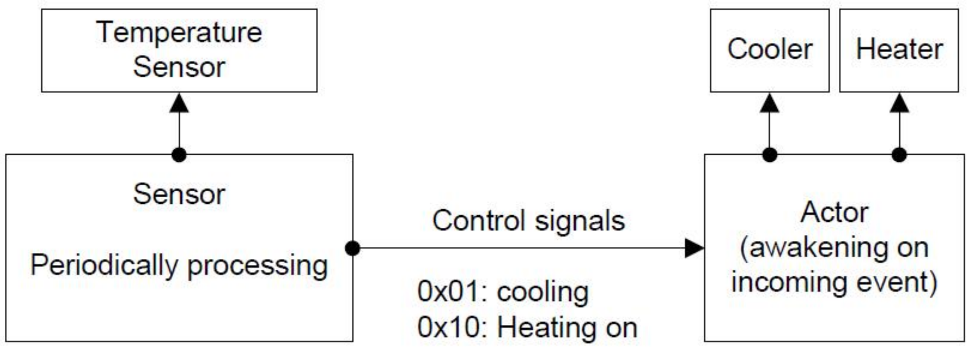

Figure 27: Software component example of regulatory circuit

Under real conditions the sensor unit and the actor unit are mostly not on the same ECU. In this case one will implement an actor and a sensor unit separately but within the same Atomic Software Component as shown in Figure 27. The sensor component can be implemented as shown in Figure 28.

Regarding the fact that the *ActorRunnableEntity* is only called if data will be received from the Sensor Components, it is not necessary to implement the case between 40 and 50 degrees. Within that range it is necessary to do exactly the same as before until the temperature comes to the opposite limit. For instance, the temperature is about 60 degrees. The cooler certainly is switched on and the heater is switched off. Encountering the barrier of 50 degrees the status will be the same. This state will firstly alter in case of 40 degrees when the cooler will switched off and the heater switched on. The same will occur vice versa. Figure 29 shows the implementation of the Actor Runnable Entity.

The following part describes the approach using dSPACE SystemDesk, which is the common system design tool for those problems in the automotive industry and evaluate it against the Software Component Generator of this thesis.

```
FUNC(void, RTE_APPL_CODE) SensorRunnableEntity(void) {
  UInt16 temperature, out;

  Rte_Call_SensorIn_TempIn(&temperature);
  if (temperature < 40) {
    out = 0x10;
    Rte_Write_SensorOut_control_out(out);
  }
  else if (temperature > 50) {
    out = 0x01;
    Rte_Write_SensorOut_control_out(out);
} }
```

Figure 28: Sensor implementation of the regular curcuit

The first part of both evaluations contains an introduction to the usage of the tools. Subsequently an overview of the particular advantages and disadvantages of the program will by given. These advantages and disadvantages will be evaluated separately at the end of this chapter.

```
FUNC(void, RTE_APPL_CODE) ActorRunnableEntity(void) {
  UInt8 control;
  Rte_Read_ActorIn_control_in(&control);
  if (control & 0x01) {
    Rte_Call_ActorOut_SignalOut(SOMETHING);
  } else {
    Rte_Call_ActorOut_SignalOut(SOMETHING);
} }
```

Figure 29: Implementation of the ActorRunnableEntity

5.1 Evaluation of dSPACE SystemDesk

The first impression of SystemDesk is very tempting as shown in Figure 30. A graphical user interface for designing software components seems very suitable and always gives an overview of used ports, interfaces, datatypes and much more.

It is possible to add new Atomic Software Components, port and interfaces as well as completely describe them. Having all elements created, the events and connections between datatypes have to be set. Afterwards SystemDesk verifies the entire project and gives warnings or even errors if the information is inconsistent, incomplete or ambiguous and exports the Software Component Description [2].

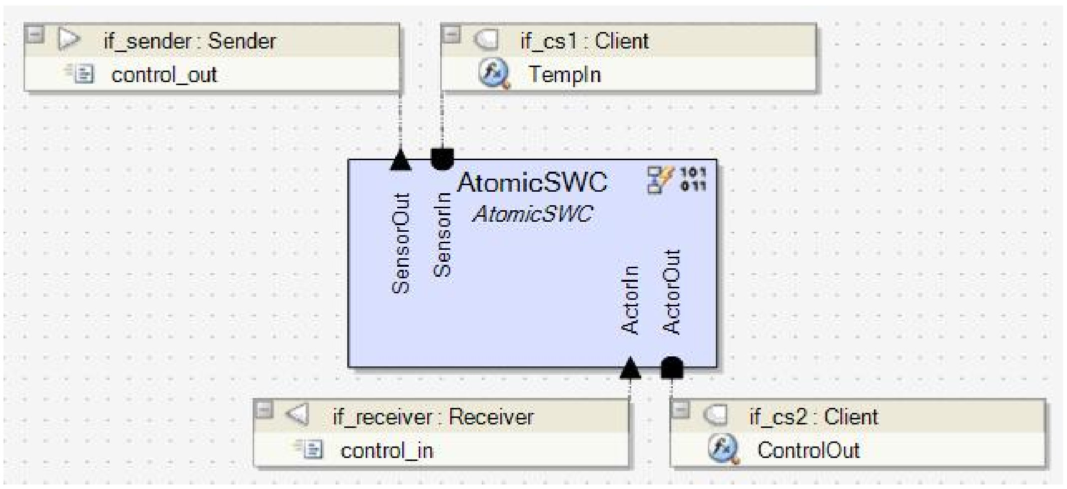

Figure 30: Ambiance of SystemDesk

After having generated the Software Component Description, it is also possible to generate the stucture of the compilable C-code. Eventually the code can be fulfilled and the project can be generated and compiled.

Conclusion

SystemDesk provides a strong environment to build not only small Software Components but to develop the whole electrical environment within the whole car and therefore to handle the complexity of the system very well. For that reason the tool is also able to optimize the system performance and help the developer to create more efficient software. Another tremendous advantage is the ability to assemble Software Components to different compositions in order to go a step further in direction of the real environment. Last but not least the strongest advantage of SystemDesk is its ability to find errors and inconsistencies in Software Components, which could lead to serious problems.

5.2 Evaluation of SWC Generator Software

The Software Component Generator does not provide any graphical user interface. However this is not necessary. To get the Software Component Description that way, there only has to be placed some additional statement within the code as shown in Figure 31.

```
//@ runnable Sensor, triggered by time 0.01, false}        // metadata
FUNC(void, RTE_APPL_CODE) SensorRunnableEntity(void) {
  UInt16 temperature, out;

  //@ interface if_cs1, datatype UInt8, TempIn             // metadata
  Rte_Call_SensorIn_TempIn(&temperature);
  if (temperature < 40) {
    out = 0x10;
    //@ interface if_sender, datatype UInt8, control_out   // metadata
    Rte_Write_SensorOut_control_out(out);
  }
  else if (temperature > 50) {
    out = 0x01;
    //@ interface if_sender, datatype UInt8, control_out   // metadata
    Rte_Write_SensorOut_control_out(out);
  }
}

//@ runnable Actor, triggered by                           // metadata
          DataReceivedEvent(ActorIn, control_in), false
FUNC(void, RTE_APPL_CODE) ActorRunnableEntity(void) {
  UInt8 control;
  //@ interface if_receiver, datatype UInt8, control_in    // metadata
  Rte_Read_ActorIn_control_in(&control);
  if (control & 0x01) {
    //@interface if\_cs2, datatype UInt8, ControlOut}}     // metadata
    Rte_Call_ActorOut_SignalOut(SOMETHING);
  } else {
    //@interface if_cs2, datatype UInt8, ControlOut        // metadata
    Rte_Call_ActorOut_SignalOut(SOMETHING);
  }
}
```

Figure 31: Complete implemenation using the Software Component Generator

As one can see, eight lines of metainformation last to place all the information that is needed. Now the code is prepared to get into the parser. By typing

```
run regulate.c
```

the parser is started and generates a file called `regulate.arxml` by default.

This file contains all used Interfaces, Ports, Atomic Software Components and Internal Behaviours as SystemDesk did before. Although it is not possible to create the Composition package due to the facts shown in the analysis chapter.

Conclusion

An enormous advantage is obviously the easy development of small test components by simply writing some lines of code which will generate the whole Software Component Description for this Atomic Software Component.

Another great advantage, considering the maintenance process later on, is the ability to see changes logged automatically by version management tools. Due to this fact it is possible for a big variety of software developers to maintain the code and its generated Software Component Description within one step and the help of one single tooling. Under this circumstances the ability to form teams with people working concurrently on the same Software Component will be highly increased.

5.3 Conclusion

Comparing the tools there evolve some advantages and disadvantages for both of them. The following criteria can be scored differently for each particular generation tool.

Teamwork Ability and Maintainability

SystemDesk itself provides the ability to seperate work tasks in packages. At first the entire structure of the Software Component has to be build and eventually the code can be generated. This fact always leads to the following conclusion: after the generation process there is no connection between the SystemDesk project and the generated C-code any more, which leads to problems of maintainability. Indeed it is not possible that more than one developer works simultaneously on the same Software Component because a Software Component can not be devided any more by SystemDesk.

The ability to work in teams is much greater for the Software Component generator. No information about the structure of the Software Components has to be evolved. For that reason it will be much easier to keep the code and the generated structure consistent due to the fact that modifying code will also modify the Software Component's structure. Any problems that may arise considering concurrent changing of code by more than one developer are exactly the same as with conventional code development. Conclusively the Software Component Generator provides more advantages in teamworking ability than SystemDesk.

Widespread of Tool Functionality

Certainly SystemDesk offers a huge amount of functionality that facilitate modelling an entire system description for the complete car environment. Due to this fact it is able to optimize the modeled system and can analyze the project on errors. With the generator all this can not be performed because the complete system information is not available (mainly the missing Compositions is the problem). Recapitulating SystemDesk certainly wins in case of comprehensive tool support because it covers the complete development demand.

Easiness of creating complete Software Component Descriptions

The first usage of SystemDesk without the help of anyone requires quite a bit of training period. However this problem will not appear after repeated utilization. For small Software Components, which need to have only some input and output interfaces and only do some calculations, it is a much faster process to create the Software Components Description using the generator. For each communication point and Runnable Entity merely one line of additional code has to be set to generate the description.

SystemDesk needs much detailed information, from which most of them actually can be extracted out of the code without much effort. The regulator software component for example took about fifteen minutes for the Software Component Description design and the code development. Of course this value depends on the experience of the developer and can only be estimated for other ones. Using the Software Component Generator the code development certainly took about five minutes too but the metainformation were only about five or six lines, which were done in about two or three minutes. Thus only about the half of the time is needed using the generator (not considering the missing composition package). If the Software Component developer aims to create small test applications, the software component generator would obviously be the better alternative.

6 Summary and Perspectives

6.1 Summary of the Generation Process

Given the problem to develop a tool that generates Software Component Descriptions from C-code leads to a variety of difficulties. The amount of needed metainformation should always be held as low as possible in order to discharge the software developer from work that he not necessarily has to do.

Consequently most of the information has to be parsed and created out of the given C-code. The intensity of information also varies from developer to developer. Hence automatic detection of datatypes could lead to malfunction or wrong information in the Software Component Description if a developer makes use of automatic typecasts from the C-compiler such that it is impossible to figure out the needed datatype.

On the one side such problems confine the functionality of the generator but on the other side the developer does not have to change his programming paradigm because it always works. Now the developer has the ability to write complete Software Components (e.i. Software Component Descriptions and C-code) much easier if he only wants to run test cases. Beyond that Software Components can be stored and managed like conventional C-code because the data for the generation of Software Component Descriptions exists as text. The source code as well as the information for the description can be managed using subversion tools like *svn*.

Last of all I want to thank Philipp Janda, scientific assistant at the University of Erlangen-Nuremberg. He offers assistance and mentioned some good points which helped a lot acquiring this thesis.

6.2 Future Work

Adding constraints that will not be checked by tresos

EB tresos does not check sematic consistency because missing paths can be added later on within another software component description.

However it would be possible to check this dependecies before the code generation step in order to improve stability. This needs to have another analysis about what has to be checked in order to not check information twice whereas the complexity can easily explode.

Integrate missing Software Component Composition

The Software Composition is the only missing package. Indeed it could be generated by a single parser that can parse a seperate file. In this file, the Composition Description could be programed in a meta language that could look like

```
Composition {
  Name: [string]
  AtomicComponentList[]
  ConnectorList[]
}
```

If the file is available, the parser could also parse this one and generate the Composition package. Otherwise if it is not available the parser will do only the work it does right now. With this information, the developer will not have to look or even write XML code himself because it would be fully generated.

Combining automatically created class container classes

Veronika Zeintl, another student currently working at EB writes a diploma thesis with the topic *Automatic generation of JAVA classes out of the AUTOSAR UML specification* . Her job is to parse the specification of used datatypes that are for example *RunnableEntity, Datatype or AtomicSoftwareComponentType.* This container classes could be applied by the generator as structures to build the information and store them. This could be an interesting topic to reduce work when the AUTOSAR version and its specification changes. The output of her implementation could be used and compiled with the generator and therefore less work is needed to be done.

Completely removing metainformation

Virtually it is possible to completely remove the metainformation. This simply leads to the following requirements:

1. The applied datatypes must be specified cleary. This means that the programmer has to avoid automatic typecasts such that the parser is able to determine the right datatype automatically.

2. Underlines within the portnames and dataelementnames must be forbidden. This is already the case in most of the guidelines but no standard. Having this requirement the name of the dataelement does not have to be given by metainformation.

3. If the parser parsed an API function which contains the name of the Runnable Entitiy, the information is given. If this does not happen, the name of the Runnable Entity is not necessary and can be set on the symbol of the Runnable Entity. The

triggering event must be placed anyway, but this would be the only remaining statement.

Having all these requirements, there exist a way to get rid of the needed interface information: create an Interface for each dataelement itself. In fact this is not what Interfaces are there for. Interfaces exist to aggregate dataelements, but it will also work if each Interface will only aggregate one dataelement. Finally metainformation for declaring Interfaces would not be necessary any longer. On the other hand this would lead to not well designed Software Component Descriptions and therefore Software Components but points out a possibility.

List of Figures

Index

Bibliography

[1] Terence Parr Dieter Frei. *ANTLR v3 documentation*. http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation.

[2] dSPACE. *SystemDesk*. http://www.dspace.de/systemdesk.

[3] AUTOSAR GbR. *AUTOSAR Schema (xsd)*. http://www.autosar.org/download/r2/AUTOSAR_Schema.zip.

[4] AUTOSAR GbR. *Software Component Template, Version 3.1.0*. http://www.autosar.org/download/specs_aktuell/AUTOSAR_SoftwareComponentTemplate.pdf.

[5] AUTOSAR GbR. *Specification of the Compiler Abstraction, Version 1.0.1*. http://www.autosar.org/download/r2/AUTOSAR_SWS_CompilerAbstraction.pdf.

[6] AUTOSAR GbR. *Specification of the RTE, Version 2.0.1*. http://www.autosar.org/download/specs_aktuell/AUTOSAR_SWS_RTE.pdf.

[7] AUTOSAR GbR. *Specification of the System Template Version 3.0.4*. http://www.autosar.org/download/specs_aktuell/AUTOSAR_SystemTemplate.pdf.

[8] Autosar GbR. *Background*. www.autosar.org, 2008.

[9] Autosar GbR. *Motivation and Goals*. www.autosar.org, 2008.

[10] Elektrobit Automotive GmbH. *AUTOSAR Training*. 2008.

[11] Ashley J.S Mills. *ANTLR Tutorial*. http://supportweb.cs.bham.ac.uk/docs/tutorial/docsystem/build/tutorials/antlr/antlr.html.

[12] Terence Parr. *AQ - Getting Started - ANTLR 3*. http://www.antlr.org/wiki/display/ANTLR3/FAQ+-+Getting+Started.