Methodology and Templates in AUTOSAR

Regina Hebig

Hasso Plattner Institut

Abstract. AUTOSAR (AUTomotive Open System Architecture) is a worldwide development partnership of car manufacturers and suppliers. The goal is to provide an open standardized software architecture for the development of automotive systems. Besides the architecture, AUTOSAR specifies templates, which define the structure of the information that needs to be stored and shared during the development of an AUTOSAR conform system. Finally AUTOSAR provides a methodology, which defines the main development activities. This paper offers an introduction to the templates and the methodology.

1 Introduction

The AUTOSAR standard defines an architecture for the development of automotive control systems [4]. This architecture consists of four layers which are described in detail in [20]. The four layers are shown in Figure 1. The uppermost layer contains the software components of the application specific part of the system, whereas the hardware aspects are located at the bottom layer. Between these two layers are the runtime environment (RTE) layer and the basic software layer, which abstract from the hardware and offer basic functionality to the software.

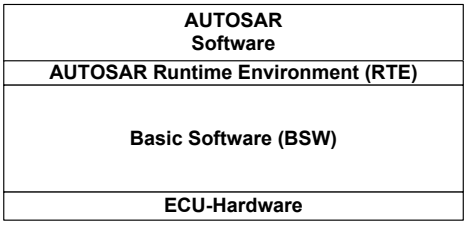

Fig. 1. The AUTOSAR architecture layers.

The software of the RTE layer as well as the basic software (BSW) on the basic software layer have to be configured for each electronic control unit (ECU) on which they are deployed. Thereby not only the characteristics of the chosen ECU but also the software parts that are deployed on the ECU have to be taken in to account.

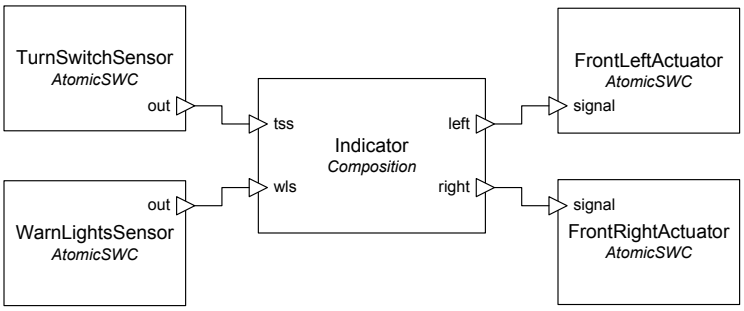

Fig. 2. The software components of the warning-light-switch example system.

An example for a system which may be built with AUTOSAR is the warning-light-switch system, which is used as an example in the System Desk Tutorial [3], too. The software part of the system consists of the components shown in Figure 2. These components are defined on the uppermost layer of the AUTOSAR architecture. The hardware of the system is shown in Figure 3. It is a car with two lights in front, two sensors (indicator switch and warning light switch) as well as three ECUs, which are connected via a CAN bus. The ECUs, sensors and the bus are described on the bottom layer of the AUTOSAR architecture.

One goal of AUTOSAR is that software and hardware can be built independent of each other as far as possible. However there is a point when software architecture and hardware structure are developed and this two parts have to be mapped together.

It is necessary to define which software components will be deployed on which ECU. For example that the 'FrontLeftActuator' component shown in Figure 2 will be mapped to the 'Front left indicator ECU' shown in Figure 3.

As the software architecture is not only developed independent of the hardware structure but also independent of the actual implementations, for each software component an appropriate implementation must be chosen. For example both the 'FrontLeftActuator' component and the 'FrontRightActuator' component can be realized by the same implementation.

In addition the communication between the software components ports must be mapped to signals and protocol data units (PDU) for the communication via the bus systems.

How the remote communication is hidden from the software components is described in [17]. In [15] is described how the basic software of the AUTOSAR architecture handles the communication of PDUs and signals.

The definition of the mappings among implementations, software components, hardware and signals is the precondition for the configuration of the RTE and the basic software for each ECU.

To support the development of AUTOSAR systems, all these activities have to be brought into an order. This order in defined by the AUTOSAR methodology [6].

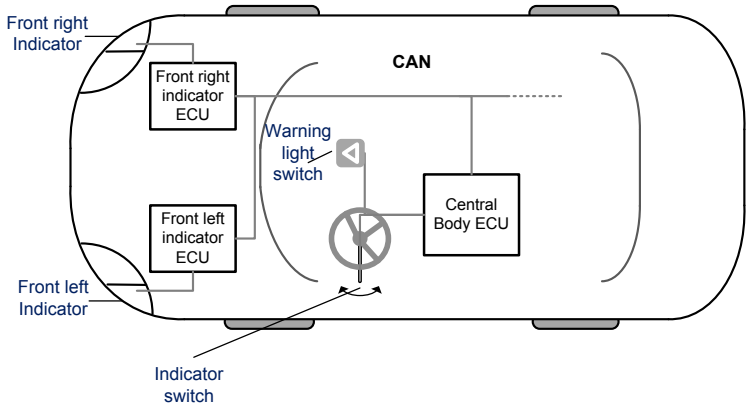

Fig. 3. The hardware structure of the warning-light-switch example system.

The single activities need input information and the results have to be documented. There may be several tools used during the development process, like for example SystemDesk. To make this possible all information must be available in a standardized format to be exchangeable. Therefore AUTOSAR provides a set of templates, which are based on an UML Profile. Each template defines a structure for the information about a specific aspect of the AUTOSAR systems description.

The AUTOSAR methodology specifies not only the flow of the activities, but also the characteristics of the input and output documents. This includes the identity of the template of which the document is an instance as well as a declaration which information are already included to the document. So the methodology defines formal which information are produced and consumed in the different activities.

The methodology is defined with the help of the Software Process Engineering meta-model (SPEM) [6]. SPEM supports the illustration of activities and of the corresponding input and output documents. In addition SPEM allows the methodology to specify how the single activities can be supported by tools.

Although the methodology defines the major steps which are necessary for the development of AUTOSAR systems it describes not a complete process. It does not define how often activities or flows of activities can be repeated. In addition the methodology defines for the most part no roles or responsibilities for the activities. However, it can be used as a basis for a development process.

AUTOSAR provides different methodologies for different tasks, but this paper only deals with the most important one. It describes the derivation of a configured executable for each ECU, starting with the description of the software and hardware. This methodology (in the following called simply methodology) consists of two parts.

The first part of the methodology starts with the template based descriptions of hardware, software and constraints over the system. The result is a description

of the whole system. This includes a mapping of software to the ECUs. The second part of the methodology describes how for a single ECU the basic software and RTE are configured and an executable is generated.

In Section 2 the templates of AUTOSAR will be introduced. Section 3 illustrates the activities of the methodology in detail.

2 AUTOSAR Templates

A template in AUTOSAR, as described in [14], is a collection of attributes, which are necessary to describe a subject of AUTOSAR. Such subjects are ECUs, software components or a whole system. Each template provides a formal definition of the information structure for its subject.

All templates are specified in UML, based on the AUTOSAR UML Profile, which is described in [5]. This profile defines a set of stereotypes for the expression of AUTOSAR concepts like ECU or hardware element. The stereotypes are structured in packages, according to their usage in the different templates.

If templates are filled with information about a subject instance they are also called AUTOSAR descriptions.

The next subsection introduces the main templates and the structure among them. After this the system template is shown in detail.

2.1 The template structure

As said above all templates are based on the AUTOSAR UML profile. The template packages in the UML Profile are related with another as shown in Figure 4. A relation between two packages show, that one package reuses concepts that are defined in the other package.

For example the *System* of the System Template owns a *SoftwareComposition*, which is defined in the Software Component Template. In the following the different templates are introduced.

The *Generic Structure* [7] defines some basic properties, which are followed by every template in AUTOSAR. An example is the fact that every element of a template has a unique identifier. For example both the concept 'CAN bus' as well as the CAN bus instance shown in Figure 3 have unique identifiers. But the Generic Structure also defines that every template starts with the element *AUTOSAR*.

The *ECU Resource Template* [12] describes the hardware resources of an ECU like the *Central Body ECU*. This includes for example information about the memory, the processing unit peripherals or sensors. The described information is relevant for the configuration of the ECU abstraction layer and the micro controller for the specific ECU.

The *Software Component Template* [9] is used to describe the software part of the system. The template includes the general description of component types with port prototypes and port interfaces. It distinguishes between composite components, which lead to a hierarchical structure of the software components

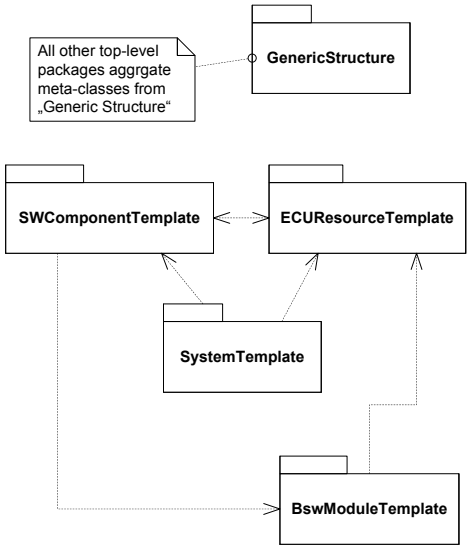

Fig. 4. Extract of the package structure of the UML Profile.

and atomic software components. They describe pieces of software, which can be mapped to ECUs. For example the Indicator software component shown in Figure 2 is modeled as a composite component.

In AUTOSAR a software component is a 'formally described piece of software existing above the AUTOSAR RTE' [9]. This means that the *Software Component Template* is used to describe entities of the *Application Layer* of AUTOSAR and AUTOSAR interfaces on the *Basic Software Layer*.

The *System Template* [13] references both the *Software Component Template* and the *ECU Resource Template*. With it this template defines the relationship between the software view of the system and the physical architecture with the ECUs. In addition, constraints over this relation can be described with the help of this template. In the next subsection this template is shown in detail.

A next template is the *Basic Software Module Description Template* [10]. It concerns the aspects of the basic software. These are all information about basic software modules and basic software cluster. The package of the BSW module description template is related with the software component template, because they both describe implementation aspects and resource consumption.

Some templates which are not shown in Figure 4 are the *ECU Configuration Template Structure* and the *ECU Configuration Parameter Definition*, which are defined in [11]. Both templates are used during the configuration of single ECUs and describe configurations, which concern basic software modules. The ECU Configuration Template Structure is exceeded by the ECU Configuration Parameter Definition. The later one includes information about restrictions of parameters and the location of their storage.

2.2 The System Template

For the better understanding of the template concept, the System Template serves as an example.

The aim of the System Template [8] is to show the relation between the software architecture, implementations and the hardware structure of the whole System. It includes the overall software architecture, a system topology for the hardware as well as several mappings and constraints. Figure 5 shows the meta model structure of the system template.

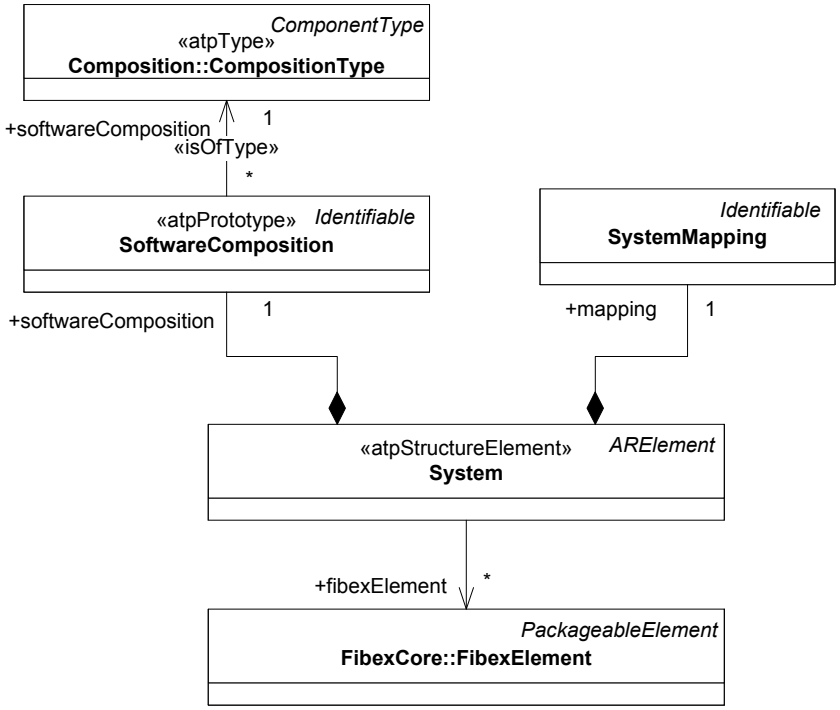

Fig. 5. The UML meta model structure of the system template.

As the meta model structure implies, the system template includes a single software composition a system mapping and references to FibexElements. Under the topic 'topology' Fibex will be introduced. First the software composition is introduced and afterward the hardware aspect, which is covert by the Fibex references. At last the system mapping is examined.

Software Composition The software composition describes the outermost software component of the described system. This is a composite component without ports. It includes the hierarchical structured software system with its

composite components, connectors and atomic components. In the example software structure (Figure 2) this outermost software component is not shown.

The description of components includes the ComponentType, PortPrototypes, PortInterfaces, DataElementPrototypes and the InternalBehavior, which is described by Runnables and port prototypes, which invoke operations. All this aspects are defined in the Software Component Template.

For example the FrontLeftActuator in Figure 2 is modeled as an atomic component with a PortPrototype, denoted with 'signal'.

Topology The referenced Fibex elements describe hardware aspects. Fibex (Field Bus Exchange Format) is an XML format for the exchange of data between tools for bus communication. It supports most of the common buses like LIN, CAN or FlexRay. Fibex allows a user to define a software architecture, a system topology to cover the hardware description and the description of communication aspects. In AUTOSAR only a subset of Fibex is used. This subset contains the topology and communication aspect descriptions.

Fibex elements like CommuniationCluster and EcuInstances describe the hardware topology. Other Fibex elements like ISignal or Frame are used to describe the communication aspects.

There is a difference between the description elements in the topology and the real hardware elements that are described with the help of the *ECU Resource Template*. An ECUInstance will be mapped to an instance of the ECU Resource Template. In the example shown in Figure 3 such ECUInstances are the *Central Body ECU* or the *Front Left Indicator ECU*, which are shown in Figure 3. An EcuInstance owns a CommunicationController. Such an element describes the communication with a CommunicationPheripheral, for example the *indicator switch*.

A further element of EcuInstance is the CommunicationConnector, which describes a bus interface and specifies a sending and receiving behavior. It will be mapped to a ECU Communication Port. This CommunicationConnector references a physical channel, which is owned by a CommunicationCluster (e.g. the CAN bus).

The CommunicationCluster describes a part of the topology, as it owns one or more physical channels, which connect communicating ECUs. However, an ECU can be connected to more than one CommunicationCluster.

Another piece of information that is hold in the System template is the communication matrix. It stores the information, which signals, frames or PDUs are sent and received on which channel at the single ECUs.

Mapping The third part of the system is the SystemMapping. It holds a set of mappings like the EcuResourceMapping, the mapping of software components to an ECU (SwCompToEcuMapping), the mapping of a software component to an implementation (SwCompToImplementationMapping) or the DataMapping.

Listing 1.1. Example SystemDesk XML plot of a SwCompToEcuMapping.

```
<SWC-TO-ECU-MAPPING
  UUID="dc6b203d-f3b8-48a1-b1bb-ebe611639946">
  <SHORT-NAME>abd8b270ad2074978a759aa82135687</SHORT-NAME>
  <COMPONENT-IREFS>
    <COMPONENT-IREF>
      <SOFTWARE-COMPOSITION-REF DEST="SOFTWARE-COMPOSITION">
        /TutorialProject/SYS_Indicator/SWA_Indicator
      </SOFTWARE-COMPOSITION-REF>
      <TARGET-COMPONENT-PROTOTYPE-REF
        DEST="COMPONENT-PROTOTYPE">
        /TutorialProject/SWA_Indicator/FrontLeftActuator
      </TARGET-COMPONENT-PROTOTYPE-REF>
    </COMPONENT-IREF>
  </COMPONENT-IREFS>
  <ECU-INSTANCE-REF DEST="ECU-INSTANCE">
    /TutorialProject/HWT_Indicator/FrontLeftIndicatorEcu
  </ECU-INSTANCE-REF>
</SWC-TO-ECU-MAPPING>
```

In addition the SystemMapping can contain MappingConstraints, which restrict the possible mappings of software components to the ECUs. This is useful, as the *System Template* is used at several stages in the development process. If the mappings are not yet defined, it is nevertheless possible to make statements about mapping aspects.

Listing 1.2. Example XML plot of a SwCompToImplementationMapping.

```
<SWC-TO-IMPL-MAPPING>
  <SHORT-NAME>ImplementationMapping2</SHORT-NAME>
  <COMPONENT-IMPLEMENTATION-REF DEST="SWC-IMPLEMENTATION">
    /TutorialProject/IMPL_TurnSwitchSensor
  </COMPONENT-IMPLEMENTATION-REF>
  <COMPONENT-IREFS>
    <COMPONENT-IREF>
      <SOFTWARE-COMPOSITION-REF DEST="SOFTWARE-COMPOSITION">
        /TutorialProject/SYS_Indicator/SWA_Indicator
      </SOFTWARE-COMPOSITION-REF>
      <TARGET-COMPONENT-PROTOTYPE-REF
        DEST="COMPONENT-PROTOTYPE">
        /TutorialProject/SWA_Indicator/TurnSwitchSensor
      </TARGET-COMPONENT-PROTOTYPE-REF>
    </COMPONENT-IREF>
  </COMPONENT-IREFS>
</SWC-TO-IMPL-MAPPING>
```

How the atomic software components are distributed across the different ECUs is specified in the Software component to ECU Mapping (*SwCompToEcuMapping*). It maps the ComponentPrototypes to the ECUs. E.g. the FrontLeftActuator component in Figure 2 is mapped to a specific ECU as shown in Listing 1.1.

The DataMapping describes primarily how the operations and data elements, which are used in the ports of the software components, are represented as signals.

The ECUResourceMapping maps the topology to the ECUResourceDescriptions. This mapping was already described under the topic topology.

The SwCompToImplementationMapping describes which implementation instantiates which ComponentPrototype. The example in Listing 1.2 shows the mapping of the implementation for the TurnSwitchSensor component.

The mappings can be restricted through MappingConstraints. The constraints can describe which components cannot be mapped to the same ECU or must be mapped to the same ECU. It is also possible to restrict which software component can be mapped to which ECU. An example for this is the constraint that the FrontLeftActuator has to be deployed at the FrontLeftIndicatorECU shown in Figure 3.

Figure 6 shows the UML meta model of the component separation and component clustering. A *ComponentSeparation* defines two components that must not be deployed at the same ECU, while a *ComponentClustering* defines a set of components that must be deployed on the same ECU.

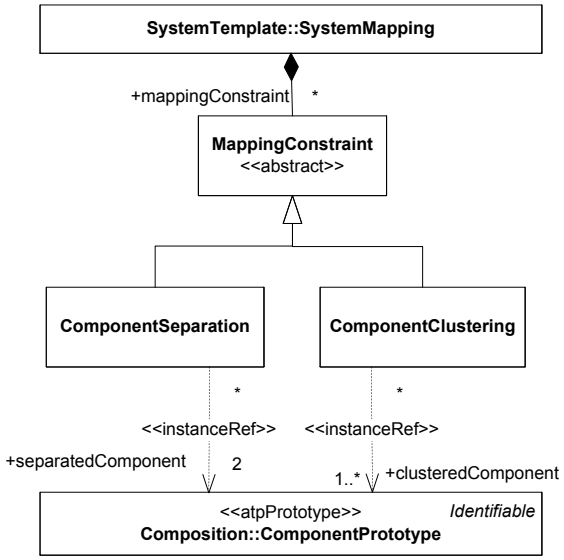

Fig. 6. UML meta model of the ComponentSeparation and ComponentClustering mapping constraints.

3 Methodology

The templates define a formal description of the structure of information. Therefore they are used to describe the work products and inputs of the activities of the methodology.

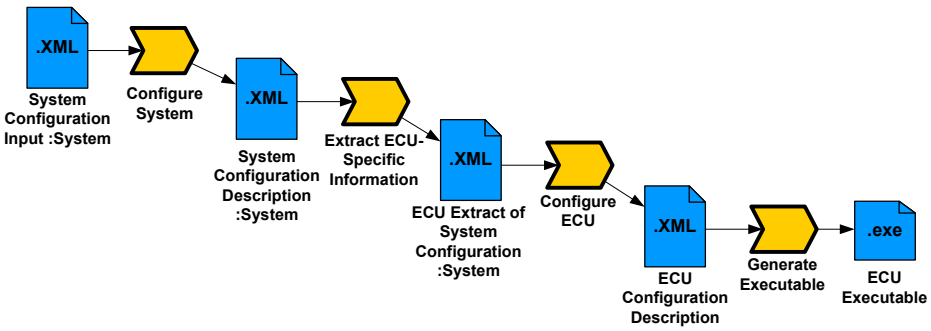

Fig. 7. The four main steps of the methodology.

As already mentioned in the introduction the methodology describes a set of activities for the development of AUTOSAR systems. Actually there is more than one 'methodology' in AUTOSAR. However, the most common methodology is the one, which leads, starting with software and hardware description, to the generation of executables for each ECU. The Sections 3.1 and 3.2 introduce the four main steps in detail.

Before the introduction of these steps, the whole extent of what is named methodology in AUTOSAR is shown. The diversity actually starts inside the 'common methodology'. There are several alternatives concerning the configuration of the ECUs or the generation of code.

In addition there is a methodology, with the aim to include measurement and calibration functionality into the system. In [18] calibration is described as the adjustment of controller software functionality according to the individual vehicle. The data that are adjusted during calibration have to be changeable until late phases of the development. AUTOSAR supports the description of the data characteristics that are affected by the calibration.

The activities that are necessary to develop an application software component and to integrate it to a system are described in the methodology for the implementation of components. The goal of this is the ability to develop the core functionality of software components independent of the rest of the system.

Finally there is a methodology for the specification of new AUTOSAR templates.

The methodology to retrieve the executables for each ECU consists of four main steps as shown in Figure 7. The first step *Configure System* concerns the whole system. The second step *Extract ECU-Specific Information* prepares the

results of the first step for the next steps, which concern only single ECUs and no longer the whole system.

The last two steps deal with the configuration of the ECU specific parts of the system (*Configure ECU*) and the generation of the executable (*Generate Executable*), as described in [11]. The next subsection shows the first and second step in detail. Section 3.2 explains the configuration of the ECU and Section 3.3 the generation of the executables.

As already mentioned the methodology also defines tools for the support of the activities. In Section 3.4 the different conceptual tools that are defined in the methodology are mapped to real tools like SystemDesk. Role responsibilities are only partially considered in the methodology. In Section 3.5 the possible roles are described.

3.1 Configuring the system

The configuration of the system has the goal to unify the description of the software components and the description of the hardware resources. The input of this activity is an instance of the System Template, which is named *System Configuration Input*. This instance references the topology of the system and with it the description of each ECU in form of an instance of the ECU Resource Description Template.

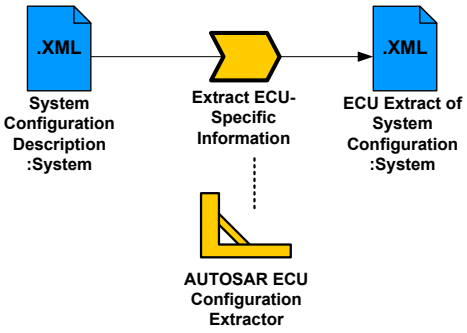

Fig. 8. Configuration of the System and extraction of the ECU specific parts.

Like the topology the top level software composition of the system is delivered with the System Configuration Input as instance of the Software Component Template.

In addition to the hardware and software aspect the System Configuration Input holds constraints, which concern the communication matrix and the mapping of hardware and software. Such a constraint is e.g. the constraint, that the 'TurnSwitchSensor' software component shown in Figure 2 has not to be deployed on the 'Front left indicator ECU' shown in Figure 3.

The second input for the activity *Configure System* is a collection of all available software component implementations. This is necessary, as the implementations are mapped to the software components in this activity. The methodology specifies requirements for a tool which supports this activity. The name of this conceptual tool is *AUTOSAR System Configuration Generator*.

As output a communication matrix and a new instance of the System Template (*System Configuration Description*) is created. The System Configuration Description references the communication matrix, the topology and the top level composition. Topology as well as top level software composition are not touched and changed during the *Configure System* activity.

The System Configuration Description includes as the most important result of this step, the new System Mapping. It brings the software architecture, the hardware descriptions and implementations together. The input and the output of this activity are instances of the System Template. While the input contains only mapping constraints, the System Configuration Description holds the actual mappings.

The Figure 8 shows the step, which follows the activity *Configure System*: *Extract ECU-Specific Information*. Also for this activity the methodology defines demands on supporting tool under the conceptual tool name *AUTOSAR ECU Configuration Extractor*. The result of the extraction is again an instance of the System Template. It is quite similar to the System Configuration Description, but it holds only the information that is relevant for a single ECU. For example there are no information about the communication between the 'TurnSwitchSensor' and the 'Indicator' software components in the ECU Extract of the 'Front right indicator ECU' shown in Figure 3.

3.2 Configuring the ECU

After the extraction of the ECU specific parts, the basic software modules have to be configured for the ECU. The configuration information for the whole ECU is described in the *ECU Configuration Description*. As shown in Figure 9 the activity *Generate Base ECU Configuration Description* instantiates the ECU Configuration Description based on the ECU extract of the System Configuration.

The *ECU Configuration Description* is in contrast to the *ECU Extract of System Configuration* not an instance of the System Template but of the *ECU Configuration Template*, which is described in [11]. During the *Generate Base ECU Configuration Description* activity the ECU Configuration Description is linked to the Basic Software Module Description of every Basic Software module that will run of the ECU and the RTE.

The RTE is handled the same way as every Basic Software module within the ECU Configuration. Therefore the RTE is also referenced, when the text is about Basic Software Modules. The Basic Software Module Description holds the information about the used implementation for a basic software module.

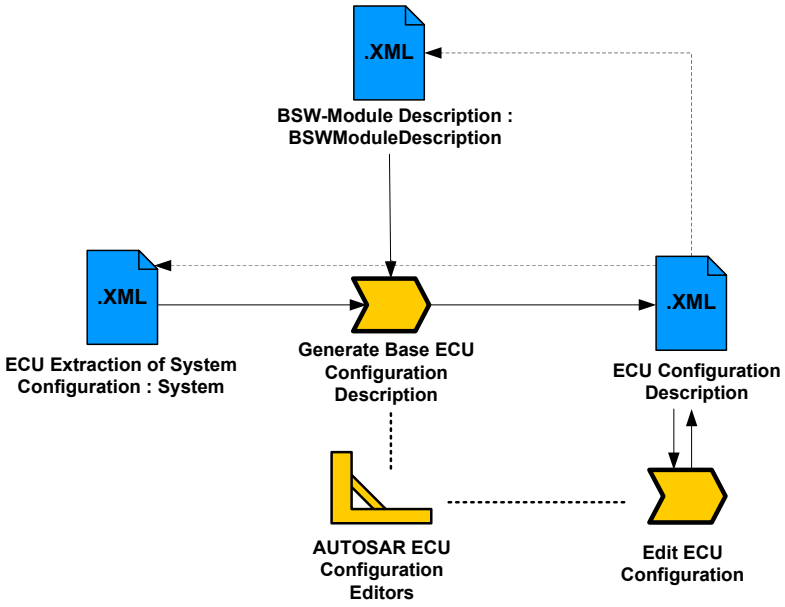

Fig. 9. Two initial steps of the ECU configuration.

In addition the *ECU Configuration Description* has a reference to the *ECU Extract of System Configuration*, as it holds the information about the software components and communication aspects.

As implied by the Figure 9 the configuration of the ECU is not terminated after the instantiation of the ECU Configuration Description. The configuration of the different Basic Software modules is done iterative in many steps. Such an activity is called *Edit ECU Configuration*. E.g. the mapping of a software component runnable 'WarnLightSensor-WlsRunnable' to the OS task 'Task_Wls' can be defined in such an Edit ECU Configuration step.

After the configuration of the ECU is fixed the activity *Generate Configured Module Code* follows. It brings the configuration information in a form, in which it can be used for the generation of the ECU executable.

3.3 Generation of the executable

Before the activity *Generate Configured Module Code* can be described, the term *Configuration Class* has to be introduced.

Each Basic Software module can be configured through several parameters. The configuration class of a parameter depends of the point in the development process of the Basic Software module, where it has to be fixed.

The deployment process of a Basic Software module contains the compilation and the linking of the code. Therefore the three configuration classes are *pre-*

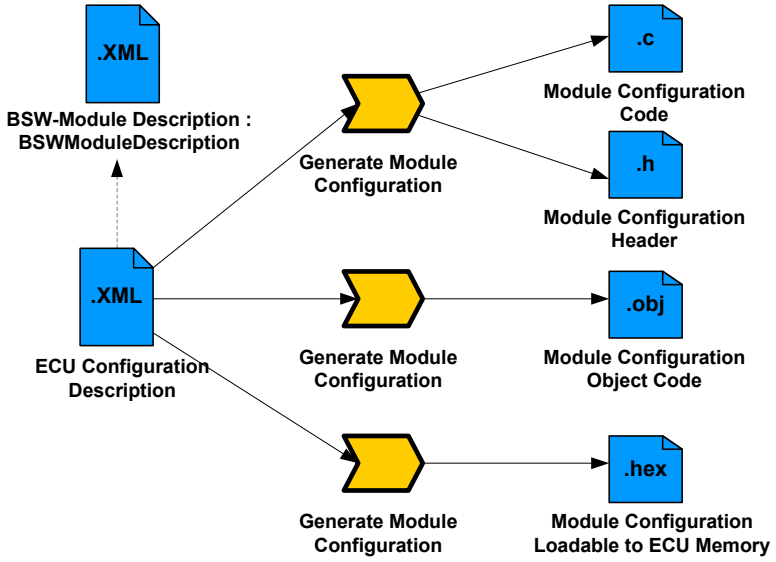

Fig. 10. Generation of the module configuration.

compile time, *link time* and *post-build time*. Parameters of the type *pre-compile time* can not be changed after the compilation, whereas parameters of the type *link time* are fixed after the linking. Parameters of the last class *post-build time* can be chosen during the boot.

For the last configuration class two alternatives exist. In the first alternative the configuration of a parameter is stored in a non executable binary file that can be downloaded to an ECU. This parameter is of the type *loadable*. The other alternative of post-build time parameters are the *selectable* parameter sets. There are multiple alternative configurations for a set of parameters that are linked into the executable.

Since the configuration classes are introduced now, the next activity of the methodology can be shown. As said above the next step after the *Edit ECU Configuration* activity is the activity *Generate Configured Module Code*, which is shown in Figure 10.

The activity is done for each Basic Software module and configuration class. There are three alternative outputs of this activity. In the first alternative the configured parameters are generated into c code and header files. As this output is not yet compiled it has to be compiled with the code of the basic software module. This variant is used for *pre-compile time* parameters.

The second alternative produces an object code file. As the object code is already compiled, this file is included to the Basic Software module during link

time. The last alternative output of the *Generate Module Configuration* activity is a binary file, which can not be executed by its own. This file can be downloaded to an ECU during the boot.

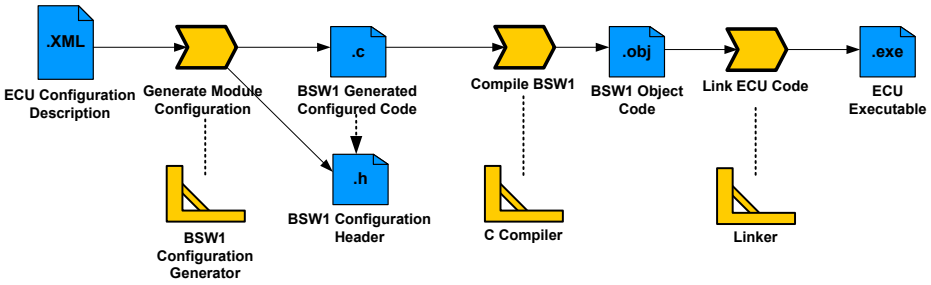

Fig. 11. Possible flow of activities for the configuration of pre-compile time parameters.

In the overview of the methodology in Figure 7 can be seen that the last step after the configuration of the ECU is the generation of the executable. The activity *Generate Module Configuration* produces different outputs, dependent on the configuration classes. That's why the *Generate Executable* activity has to deal with the different configuration classes, too.

There are different alternative approaches for the generation, for each configuration class. In the following one approach for each configuration class is introduces.

Figure 11 shows the activities for a Basic Software module (named BSW1) with *pre-compile time* parameters. In this variant the *Generate Module Configuration* activity generates not only the source code for the configured parameters, but generates the whole Basic Software module source code with the configured parameters inside.

After this step the code is compiled and linked to the executable. An example use case for parameters that are fixed after the compilation is the enabling of a macro for error tracing during the development.

Figure 12 shows the generation steps for *post-build time* selectable parameters. Source code for the different configuration sets is generated. The source code of the alternative configuration sets is then compiled into one object code file. Meanwhile the Basic Software module is compiled without configuration. With the linking the two object codes came together within the executable.

As there are now alternative sets of configuration data within the executable, the choice which set is used can be made after the link time. For *link time* parameters a very similar procedure can be chosen. Therefore the number of alternative configuration sets has to be one. An example use case for a *link time* parameter is the setting of a unique channel identifier.

The Figure 13 shows the procedure for loadable *post-build time* parameters. As visible at the bottom of the picture, the code of the Basic Software module

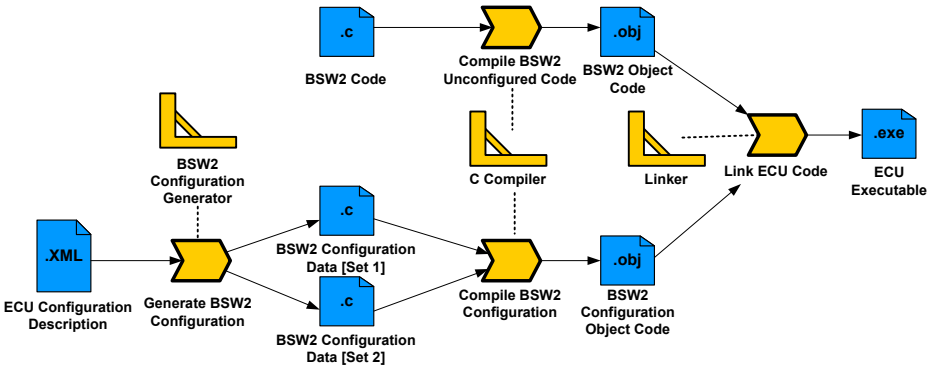

Fig. 12. Possible flow of activities for the configuration of selectable post-build time parameters.

is compiled and linked without configuration. Meanwhile the binary file is generated direct out of the ECU Configuration Description. This direct generation requires a tool that is capable to generate the binary file directly. There are alternative approaches, which split this activity into several steps.

In the result the executable refers to the loadable binary file. As the binary file can be downloaded at boot time, the choice of the configuration is located after link time.

3.4 Tools in the methodology

As mentioned above, the methodology defines requirements on tools that support of the different activities. In [19] the tool SystemDesk is introduced as an AUTOSAR supporting tool. The aspect of simulation of software components and their interactions within SystemDesk and TargetLink is captured in [16].

Did the two tools capture the role of some of the tools, which are defined in the methodology? TargetLink seems to do not. It enables the generation of code out of MATLAB, Simulink and Stateflow [2] and supports AUTOSAR with the *TargetLink AUTOSAR Block Library*. Therefore it can be used to model the behavior of the Runnables, based on the other tools and enables the generation of c code. However, it seems not to support one of the methodology activities directly.

In contrast SystemDesk seems to satisfy the requirements of several of the 'AUTOSAR tools'. SystemDesk, as it is defined for the *AUTOSAR System Configuration Generator*, supports the creation of the system mappings and the communication matrix. SystemDesk supports only a restricted set of ECUs. However, for this set the complete automatic extraction of the ECU specific part out of the System Configuration is possible. This is the task of the *AUTOSAR ECU Configuration Extractor*.

Finally the *AUTOSAR ECU Configuration Editor*, which is defined to support the ECU specific configuration, seems to be only partially supported. Sys-

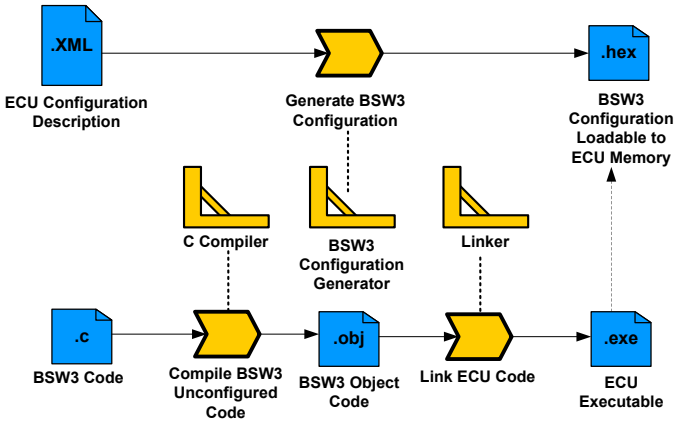

Fig. 13. Possible flow of activities for the configuration of loadable post-build time parameters.

temDesk can generate an instance of the ECU Parameter Configuration Template. However, many requirements for this tool are not satisfied. An example is that SystemDesk seems not to enable simple merges between ECU Configuration Descriptions.

An example for a tool with explicit support for the ECU configuration and the generation of code is the *EB tresos Studio* [1]. It has the explicit goal to support the configuration workflow of the AUTOSAR methodology.

3.5 Roles in the methodology

As already mention the methodology defines no roles for the most activities. Nevertheless in [11] the methodology specifies, which roles are responsible for the configuration of the basic software for the ECUs.

There are three roles that a participant in the development process of a car can play. The first role is the OEM, who is the car manufacturer. The other two roles are Tier-1 supplier and Tier-2 supplier. A Tier-1 supplier builds subsystems of the car, while a Tier-2 supplier delivers only parts of the subsystems. Actually there are also the Tool-vendors. However, they are not concerned by the activities of the methodology.

The methodology defines role responsibilities for two activities. These activities are *Edit ECU Configuration* and *Generate Module Configuration*. The configuration classes, which we introduced in Section 3.3, are crucial for these role responsibilities. Pre-compile time parameters have to be configured by the Tier-2 supplier, while link time parameters have to be configured either by the

Tier-2 or the Tier-1 supplier. Finally the post-build time parameters are configured by the OEM.

A role that is responsible for the configuration of one parameter type is also responsible for the corresponding activity *Generate Module Configuration*. The Tier-2 supplier, which configures the pre-compile time parameters, is also responsible for the generation of the configuration code. The object code for the configuration of link time parameters is generated by the responsible Tier-1 or Tier-2 supplier. The OEM is responsible for the generation of the loadable binary files.

The tendency is that the producer of a system part is also responsible for the configuration of this part. What is not yet defined in the methodology is the responsibility for other activities like *Generate Base ECU Configuration Description*, *Extract ECU-Specific Information* or the activity *Configure System*. All three activities concern system parts that are not in the single responsibility of one supplier.

It would be reasonable that the OEM is responsible for the activity *Configure System*. Generally there is always one role, which holds the responsibility for a system part. For example an ECU may be in the responsibility of a Tier-1 supplier or alternatively in the responsibility of the OEM. It would be logical that this responsible participant is also responsible for associated activities.

4 Conclusion

The templates in AUTOSAR standardize the information structure and thus enable the information exchange between different tools. The methodology defines the activities of the development process and with it the point in the development process, where a work product has to be available is defined. Thereby the cooperation among the different manufacturers and suppliers becomes more standardized.

There is still standardization work to do concerning the methodology and templates. For example AUTOSAR just started to consider roles like OEM or supplier within the methodology, as explained in Section 3.5. However, with the templates and the methodology AUTOSAR offers a basis for a standardized cooperation between the different participants of the development process of an automotive system.

References

[1] EB tresos: The Product Family for the Development of ECU Software. Web site. Found on `http://www.elektrobit.com/what_we_deliver/automotive_software/products/eb_tresos_-_ecu_software_development/eb_tresos_studio`.

[2] TargetLink. Web site. Found on `http://www.dspace.de/ww/de/gmb/home/products/sw/pcgs/targetli.cfm?nv=n2`.

[3] dSPACE. *System Desk Tutorial*, 2008.

[4] Simon Frst. Autosar - an open standardized software architecture for the automotive industry. 1st AUTOSAR Open Conference, 2008.
[5] AUTOSAR GbR. UML Profile for AUTOSAR V1.0.1, 2006.
[6] AUTOSAR GbR. AUTOSAR Methodology V1.2.2, 2008.
[7] AUTOSAR GbR. Generic Structure Template V2.1.2, 2008.
[8] AUTOSAR GbR. Requirements on System Template V2.1.2, 2008.
[9] AUTOSAR GbR. Software Component Template V3.1.0, 2008.
[10] AUTOSAR GbR. Specification of BSW Module Description Template V1.1.0, 2008.
[11] AUTOSAR GbR. Specification of ECU Configuration V2.0.2, 2008.
[12] AUTOSAR GbR. Specification of the ECU Resource Template V1.0.5, 2008.
[13] AUTOSAR GbR. Specification of the System Template V3.0.4, 2008.
[14] AUTOSAR GbR. Template UML Profile and Modeling Guide V2.2.2, 2008.
[15] Johannes Gosda. AUTOSAR Communication Stack, 2009.
[16] Alexander Krasnogolowy. Simulation of Automotive Systems in the Context of AUTOSAR, 2009.
[17] Nico Naumann. Runtime Environment & Virtual Function Bus, 2009.
[18] Jrg Schuffele and Zurawka Thomas. *Automotive Software Engineering.* ATZ/MTZ-Fachbuch, 2006.
[19] Sebastian Waetzoldt. Modeling ans Development of AUTOSAR using SystemDesk, 2009.
[20] Robert Warschofsky. AUTOSAR Software Architecture and Application Layer, 2009.