Automated Configuration of Time-Critical Multi-Configuration AUTOSAR Systems

Master Thesis

Submitted in Fulfilment of the Requirements for the Academic Degree M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by:  Kunal Chandmare
Student ID:  401441
Date:  12.07.2017

Internal Supervisor:  Prof. Dr. W. Hardt
Dipl.-Inf. Renè Bergelt

External Supervisor:  Philipp Schleiss, M.Sc.
(Fraunhofer ESK)
Abstract

The vision of automated driving demands a highly available system, especially in safety-critical functionalities. In automated driving when a driver is not binding to be a part of the control loop, the system needs to be operational even after failure of a critical component until driver regain the control of vehicle. In pursuit of such a fail-operational behavior, the developed design process with software redundancy in contrast to conventional dedicated backup requires the support of automatic configurator for scheduling relevant parameters to ensure real-time behavior of the system. Multiple implementation methods are introduced to provide an automatic service which also considers task criticality before assigning task to the processor. Also, a generic method is developed to generate adaptation plans automatically for an already monitoring and reconfiguration service to handle fault occurring environment.

Keywords: mixed criticality, automatic generation, run-time adaptation plans, AUTOSAR, tooling
Contents

List of Figures ................................................. 5
List of Tables .................................................. 7
List of Abbreviations .......................................... 8

1 Introduction .................................................... 9
  1.1 Motivation ............................................... 9
  1.2 Problem Statement ..................................... 10
  1.3 Thesis Structure ....................................... 12

2 State of the Art ............................................... 13
  2.1 Scheduling Synthesis .................................. 13
    2.1.1 Real-Time Tasks Allocation ....................... 13
    2.1.2 Mixed Criticality ................................ 14
  2.2 Runtime Reconfiguration ............................... 14
    2.2.1 Graceful Degradation ............................. 16
    2.2.2 SAPC Software Architecture ..................... 17
  2.3 AUTOSAR Tooling ...................................... 19

3 Technology .................................................... 21
  3.1 AUTOSAR ............................................... 21
    3.1.1 Layered Software Architecture .................. 21
    3.1.2 Design-Tool Work Flow (Methodology) .......... 28
    3.1.3 ECU Module Definitions .......................... 29
    3.1.4 Timing Extensions ................................ 31
  3.2 Technical Background ................................ 33

4 Concept ....................................................... 36
  4.1 Overview of Design Flow Integration ................ 36
    4.1.1 Application Design (System Modeling) ......... 36
    4.1.2 Architecture (Operational Modes Environment) 37
    4.1.3 Planning (Scheduling) ............................ 38
    4.1.4 Configuration .................................... 39
    4.1.5 SAPC .............................................. 40
4.2 Scheduling of Mixed Critical Runnable .......................... 41
4.3 Algorithm For Hierarchical Reconfiguration Plans ............. 46

5 Implementation .................................................. 48
  5.1 Data Accumulation ........................................ 48
  5.2 Partitioning For Task Allocation ............................ 49
  5.3 Automatic Module Generation ................................ 55
    5.3.1 OS .................................................. 55
    5.3.2 RTE ................................................ 59
  5.4 Generation of Generic Reconfiguration Plans ................... 61

6 Evaluation ....................................................... 68
  6.1 Setup ..................................................... 68
  6.2 Case Study ................................................ 68
    6.2.1 System Architecture .................................... 69
    6.2.2 Test Bench ............................................ 70
  6.3 Fail-Over Time Analysis ..................................... 71
  6.4 Discussion ................................................ 72

7 Conclusion and Outlook ........................................ 74

Bibliography ...................................................... 76
# List of Figures

1.1 Design Process on V-Model ........................................ 11

2.1 SafeAdapt Corners [38] ............................................. 15
2.2 Graceful Degradation .............................................. 17
2.3 Degraded Instances [37] ........................................... 17
2.4 SAPC Software Structure .......................................... 18

3.1 AUTOSAR Architecture ............................................ 22
3.2 VFB To RTE Mapping [35] ....................................... 23
3.4 OSEK Task State Model ............................................ 26
3.5 Schedule Table State Model [10] ................................. 27
3.6 Structure Of Schedule Table[10] ................................. 28
3.7 AUTOSAR Methodology [48] ...................................... 29
3.8 UML Diagram For Definitions [30] .............................. 30
3.9 ECU Configuration File - Tool-View ............................. 31
3.10 Concrete Pattern Event Triggering Periodic Repetition[13] ... 32

4.1 Design Work Flow [19] .............................................. 36
4.2 System Mode View .................................................. 37
4.3 Timing View ......................................................... 38
4.4 Result View ........................................................ 39
4.5 Block Diagram for Automated Configuration .................... 40
4.6 OS Task Runnable Execution ...................................... 41
4.7 Runnable Conflict Representation ................................ 42
4.8 Greedy Graph Coloring ............................................ 43
4.9 Greedy Analysis .................................................... 44
4.10 Generic Adaptation Tree .......................................... 46
4.11 Breadth Search [4] .................................................. 47

5.1 Data Extraction Links .............................................. 49
5.2 Configuration Tool-View .......................................... 56
5.3 Adaptation Algorithm Flow ....................................... 62

6.1 Demo-Car System Model .......................................... 68
6.2 OS Module .......................................................... 69
6.3 RTE Module .......................................................... 70
LIST OF FIGURES

6.4  Hardware Setup[19] ........................................... 71
6.5  Fail-Over Analysis ........................................... 72
List of Tables

3.1 Semantics of cseCodeType [13] ......................................... 33
List of Abbreviations

FEV  Fully Electric Vehicle
ECU  Electronic Control Unit
RTE  Run-Time Environment
OS   Operating System
ASIL  Automotive Safety Integrity Level
SAPC  Safe Adaption Platform Core
AUTOSAR  Automotive Open System Architecture
MCS  Mixed Criticality System
RMS  Rate Monotonic Scheduling
DM   Deadline Monotonic
EDF  Earliest Deadline First
MCAL  Microcontroller Abstraction Layer
UML  Unified Modelling Language
SWC  Software Component
ISO  International Organization for Standardization
VFB  Virtual Functional Bus
BB   Basic Block
EP   Expiry point
SSM  Scheduled System Model
ST   Schedule Table
SR   Scheduling Requirements
AM   Adjacency Matrix
AP   Adaptation Plan
FIFO  First In First Out
ARTOP  AUTOSAR Tool Platform
RCG  Runnable Conflict Graph
CAN  Controller Area Network
1 Introduction

1.1 Motivation

In the current state of the automotive domain, there is a huge demand for systems with high availability. This is mostly due to the evolution of autonomous driving, in which the driver is kept partially or completely out of the control loop. For example, on the occasion of critical component failure when the driver is not in control of the vehicle, the system should be operational for a short span of time with a degraded version of the failed component until the driver regains control of vehicle. Therefore to ensure safety, functions with high criticality (e.g. X-by-Wire) need a mature redundancy solution. To address the need for safety, reliability and cost efficiency in future automated vehicles, there is a need for the development of a novel architecture to manage complexity through generic, adaptive, and system-wide fault handling. Initially, mechanical redundancy mechanisms were able to address the risk of electronic system failure but having hardware redundancy affects the non-functional properties of a vehicle in large extent. Nevertheless, dual or triple redundancy has been accepted in the Aerospace industry because such investment is justifiable considering the fact that a safety-critical function has to be fully fail-operational. Whereas, an unit cost industry such as the automotive domain is less demanding for fully-operational behavior and this is totally reasoned on the fact that an aircraft cannot be stopped in mid air. Still, to make sure that a vehicle stops safely after deviation of the system from normal behavior, a downsized set of critical functions has to be operational. Therefore, adaptation poses as a viable solution to increase the reliability of highly safety-critical applications without relying on mechanical fall-back solutions [38]. Hence, a holistic approach is taken to handle system-wide fault where a platform encapsulates basic adaptation plans to relocate and update functionalities to achieve fail-operational behavior using static re-configuration with a fixed set of modes that build on static safety validation. These set of modes are different system operation modes where each mode is configured for a unique failure situation.

Being creatures of comfort and convenience, there are demands for new features and functions in the cars. As these features are enabled by electronics, the value of electronics in a car is increasing rapidly. As a result, potential number of control units in a car has increased. Considering the increasing number of ECUs even in a basic vehicle model, the possible number of modes of the system can even surpass thousand in numbers. Since each mode has a different environment, the
configuration of system scheduling parameters for each mode can be difficult to manage manually and is also admissible to human error.

Essential parameters of system configuration are classified and encapsulated in different modules in present middleware frameworks such as AUTOSAR. Since the AUTOSAR standard already focuses on interoperability between devices through standardized communication interfaces, it is well suited to be further evolved into a system architecture capable of seamlessly exchanging functionality between different platforms. Hence, it enables the integration on any kind of AUTOSAR-based platform. The current approach to configure system parameters is manual using some configuration tools such as Arctic Studio. This conventional approach towards re-configuration of multiple parameters for thousands of modes is not feasible, susceptible to human errors and inefficient; hence there is a need to automate the generation of configuration parameter values. Automation of such a configuration process can be efficient and removes the occurrence of human errors.

Also, a platform with a monitoring and re-configuration service needs a generic approach to generate adaptation plans to handle transition of a system from one operation mode to another which eventually will make this platform more feasible with a different system model having a number of computing units.

1.2 Problem Statement

Development of an automotive embedded system faces many challenges due to the increasing complexity of Electric or Electronic (E/E) architecture and a short period for product development. The challenge, however, is to ensure consistency of the concept constraints and configurations [33] during the entire product life cycle. Current automotive system architecture (AUTOSAR), improves the re-usability of different functionalities to different manufacturers. However, such a facility creates a complex system architecture making application development independent of the hardware. Hence, to create the required interface between application and hardware in the later stage of development, different modules were introduced. The parameters of these modules must be configured in order to achieve a complete system.

In order to achieve fail-operational behavior in the automotive system, a system model is composed with the information of possible failure modes. Due to different composition, each mode requires different schedules. So, to keep the system on the scale of given run-time requirements in every system mode, scheduling related parameters are need to be configured for each mode on each distributed computing unit. These parameters are encapsulated in different modules for currently accepted automotive framework AUTOSAR. These modules are nothing but Run-Time Environment (RTE) and Operating System (OS) module, which are configured to develop an individual ECU. As shown in Figure 1.1, during development of system
in the left arm of V-model (automotive system designing model), the design process is complete till system planning but there is a gap between a planned system and ECU development. A conventional approach to fill the gap is not feasible because it is difficult to handle and admissible to human error as discussed in Section 1.1.

During configuration of OS and RTE module, there is an AUTOSAR specific concept called “Runnable to Task mapping”. This basically assigns software functionality wrapped around the run-time entity called “Runnable” (detail in 4.2) to the processor tasks. Generally, this is a part of process which guarantees system schedulability, so that grouping of runnables to same task should not affect the performance and real-time behavior of system. Therefore, there are several scheduling algorithms [34, 27, 51] to counteract this problem. Whereas in the context of the thesis, the grouping of runnables to the task is considered for the planned system model. If the mixed critical (according to ISO 26262, criticality levels are termed as Automotive Safety Integrity Level (ASIL)) runnables are assigned to the same processor tasks then there is a possibility of lower critical function interfere the execution of safety critical function. Also, these grouped runnables shares the same memory being on the same task. Hence, a memory corruption from less critical application might affect the behavior of the critical function. Therefore, there is a need of an approach which considers ASIL level of applications and makes grouping of runnable to the tasks, such that there is no deviation from required real-time execution of a system.
An existing platform, which was developed during previous research [38] named Safe Adaption Platform Core (SAPC) has adaptation and re-configuration plans while monitoring and managing availability of application during runtime (explained in Section 2.2.2). This platform is not fully integrated into AUTOSAR architecture. Also, SAPC is only compatible with particular system model examples and supports only manual configuration. Such a manual method to configure an adaptation plan for each possible failure mode is again difficult and admissible to the human error.

1.3 Thesis Structure

This thesis is divided into six chapters, where Chapter 2 gives an idea about previous research on the relevant topic, current state of work and how it was helpful to create a base for the research of this thesis. Tools and technology used during implementation of the thesis work are introduced and explained in detail during Chapter 3. Whereas Chapter 4 elaborates on the concrete concept with overview of present design work flow, based on which this thesis work is developed. Furthermore, Chapter 5 introduces the algorithm, which is written and explained in detail to fill the gap in present developed system as discussed in Chapter 1.1 and 1.2. Finally, generated system from the work of this thesis is evaluated on the test set-up for its performance, and the possible future work discussed in Chapter 6.
2 State of the Art

A design process whose objective is to achieve fail-operational behavior by means of system re-configuration during the runtime, as discussed in 1.2, requires a tooling extension of automatic configurator to configure scheduling relevant parameter of the system. From a single computing unit perspective, this configurator is utilized to configure parameters from centrally scheduled system modes to ensure the real-time behavior of system. Hence, research has been done to understand different aspects of schedule synthesis in automotive embedded system relevant to this thesis. Furthermore, the evolution of runtime re-configuration which uses compositional system modes to maintain availability of functions is discussed. Also, state of the art for this generic runtime re-configuration platform is explained in detail. Supporting such design flow with automatic plug-ins requires well programmed tooling on the exchange files. Since the design process is based on the AUTOSAR framework, some previous AUTOSAR-based tooling papers are mentioned which helped to create base for the implementation of this thesis.

2.1 Scheduling Synthesis

Within the automotive industry, schedule configuration is typically only performed at the control unit level and at most synchronized to a bus system, such as FlexRay or Ethernet. An algorithm for schedulability analysis [27] is provided for non-preempting task with constraint of periodicity. Similarly, some research paper [46, 36, 22] explains scheduling model based on task allocation to distributed control unit whereas, [44] synthesizes schedules under real-time constraint on communication bus. These research states that the allocation method of real-time tasks to computing unit has abundant effect on the schedulability of a system.

2.1.1 Real-Time Tasks Allocation

Many studies have been carried out for static real-time application allocation on processors [34] but only few [27, 40, 42] have taken preexisting constraint (such as periodic, arbitrary) among tasks in consideration and explored different techniques for static assignment of tasks to processors. Later on, mapping methodology evolved in embedded real-time systems, where functional tasks are assigned to the processor to ensure system's required behavior [52]. However, these approaches cannot be used directly in the automotive system architecture such as AUTOSAR, where there is another level of mapping runnables to the task set. There are several studies dealing
with runnable to the task mapping [41, 31, 54] but few [51, 20] share the common base for the research of this thesis. Further, heuristics [28, 15] were developed to automate such a mapping to reduce the number of tasks which is one of the objective of this research, too. However, [28, 15] do not consider the criticality of the application before grouping them together. The requirement of this project is to make runnable partitioning in a way that it is compatible with already planned schedules for each mode considering their criticality level.

### 2.1.2 Mixed Criticality

Integrating different level of critical application component on to the same hardware platform is one of the tendencies in design of real time embedded systems such as automotive systems, which is called Mixed Criticality System (MCS). Therefore, development of a system must guarantee that error in low critical application does not affect the execution of higher critical software part which is called “Criticality Inversion”. In the automotive domain, scheduling is typically priority driven, therefore safety standards like ISO 26262 require freedom of interference for each criticality level, which gets violated in case of criticality inversion. The first paper on MCS was published by Vestal[49]. Later [17] proved that Audsley approach [8] of priority assignment was optimal. Whereas, [14, 23] developed a system scheduling model restricting system criticality levels to two (LO and HI). Extension to this model [23] allowed involvement of up to five criticality levels as found in the automotive industry. Automotive software standards such as AUTOSAR also address mixed criticality issues as it is recognized to be supported on their platform. Continuing on the result of the previous papers [46, 36, 22] research paper [16] explores a certain timing protection mechanism to handle mixed criticality. Also [43] gives an approach for integration of mixed critical application on the same control unit. All of these research papers concentrate on handling mixed criticality before scheduling system. In contrast to the conventional way, that paper attempts to handle functional mixed criticality of an already scheduled system.

### 2.2 Runtime Reconfiguration

Several research projects with the objective of dynamic adaptive embedded systems by means of runtime re-configuration were developed. Out of these, HEMIS [3] project develops a health monitoring system which provide fail-safe state for electric power train. DySCAS [20, 6] project for dynamic reconfiguration in automobile with middleware support proposed with possible problem[21] during re-configuration. Based on iLand [37] middleware, architecture [24] is developed supporting service based reconfiguration in networked embedded system. However, the result of these projects do not support current automotive standards such as AUTOSAR and ISO26262 [38]. The recent SafeAdapt European Project, which aims to obtain fail operational behavior in future fully electric vehicles (FEVs) [38], has
developed “Monitoring and Reconfiguration Service” [37] using generated schedules for system operational modes.

In context of risk management, a failure mode and effect analysis is introduced in paper [5], where system modes are brought to manage failure of a component. Analyzing different possible failure modes manually is laborious, therefore [25, 39] introduced tools to generate failure mode information automatically. Further, these system failure modes are utilized with simulation for assessment of safety critical real-time control systems embedded in automobiles [50, 47]. Whereas [29] introduced component modes with semantic context information to provide different scheduling or control strategy. Recently, SafeAdapt [19, 38] project utilized planned system mode compositions to ensure availability related safety properties.

As mentioned in Section 1.2, SafeAdapt is a platform which provides monitoring and re-configuration service [38]. A platform core encapsulating the basic adaptation mechanisms for relocating and updating functionalities is developed on basis of AUTOSAR. This part explains some related concept and existing working model architecture of Safe Adaptation.

![Figure 2.1: SafeAdapt Corners [38]](image)

Traditionally fail-operational behavior in vehicle industry is achieved through dedicated redundancy with confederate architecture. In contrast, being unit-cost
industry, the automotive domain desires a solution which has limited number of spare resources. This is proposed by recent research on graceful degradation (detailed explanation in 2.2.1). Here, future automated vehicles will only require a set of critical applications to be available for a short amount of time. With respect to limited resources, availability is ensured by developing a generic monitoring and reconfiguration service through SAPC module on the notion of Safety-Element-out-of-Context [45]. During run-time, predefined properties of the system must be preserved by the run-time control to guarantee the quality of safety-critical applications and prevent unwanted or uncontrolled behavior. In return, the system is able to gracefully handle unforeseen situations and guarantee the predefined quality of safety-critical applications. For the realization of such kind of run-time control in vehicles, the mode management mechanisms specified in the AUTOSAR standard are utilized and enhanced. This platform provides scalable methods and techniques for controlled adaptation and reconfiguration depending upon the current mode of a system, which is stated from unavailable heterogeneous computation unit resource. Each mode represent system environment on failure of one or more Electronic Control Unit (ECU)s in the whole automotive system. Based on the concept of application mobility, more elaborate strategies for adaptation are realized to empower the E/E system to reconfigure itself according to the needs of each mode.

There are some aspects which are considered while developing such an ambiguous platform. As shown in Figure 2.1, reliable technique of controlling adaption of the system during run-time maintains the system's functional safety, availability and predictability of system behavior. Obedience to develop good quality predefined safety-critical application during design process supports controlled run-time adaptation. To make these adaptation and function reallocation practically feasible and according to automotive safety standard regulation, such as ISO 26262, is also considered for safety assessment during its development. Since monitoring and reconfiguration service (MRS) concept is based on graceful degradation, the next section introduces the concept of graceful degradation.

### 2.2.1 Graceful Degradation

In overload condition, it is not always recommended to remove the functionality completely but instead to have a degraded version of the function in each system mode [37, 38]. Therefore, system models are capturing fine-grained information on resource requirements by allowing compositions to be assigned to system modes and functionalities arranged by their importance. This information is configured into system model during architecture stage of the design process, which is introduced in the next section.

Basically, when one of the resources (considering resource as a computational unit) fails, then a degraded instance of the critical functional component is remapped to
2 State of the Art

Figure 2.2: Graceful Degradation

the available resource to keep it operational in a system. In Figure 2.2, there are two control units, where both have one critical and one comfort function. When control unit 1 stops its normal mode behavior, less important functionality, such as comfort in this case, is deactivated and replaced by a degraded instance of critical function from control unit 1. As shown in Figure 2.3, comfort functionality has less importance than steering. The system modes are defined as the system representation of its environment on failure of a computation resource. Whereas, functionalities are modeled with several different modes of a composition that is admitted in mutual exclusion manner that degrades the function internally (e.g. Normal and Degraded Mode for Automated Driving in Figure 2.3).

Figure 2.3: Degraded Instances [37]

2.2.2 SAPC Software Architecture

This section is more specific about the software architecture of the SAPC platform. In this platform, synchronously operating software instances of SAPC are mapped to ECUs which manage the different system modes. The execution period of such software components of SAPC is based on fail-over time of managing functionalities, which is nothing but the maximum amount of time a functionality remains unavailable. SAPC is based on reporting and evaluation concept where it is responsible for broadcasting (“Local Failure Handling”) health of managing application in state of heart-bit eventually forming “Global System State” as shown in figure 2.4 and evaluating this system state to trigger reconfiguration. To evaluate such global system state, one look up component is deployed on each ECU in which the states of all application in system is integrated. Based on this component “Planning Fail-
ure Handling” as shown in figure 2.4, a lookup occurs which either keeps the control unit in the same state or reconfigures it using previously known “Adaptation Plans”. Even though the term used for the reconfiguration plans is adaptation, it do not implicate generic dynamic adaptation. Since the design process is developed in accordance with ISO 26262 [38] which prohibits dynamic reconfiguration and promotes more static or deterministic planning. So when the term Adaptation Plan is used in this thesis, it is in context of deterministic reconfiguration plans for the system to maintain the predictability as it is used in figure 2.1. An Adaptation Plan has all application instances mapped on control unit and a new schedule for it. An essential generic algorithm, which generates such an Adaptation Plan automatically for any system model, is developed in this thesis work.

![SAPC Software Structure](image)

**Figure 2.4: SAPC Software Structure**

Prerequisites for automatic adaptation plan generator are well defined system modes during development of the system model and planned system schedules for each system mode. So, this algorithm uses the same technique as previous adaptation plan creation but making it more automatic and generic such that it is compatible with any system model rather than specific and manual. Therefore, there are
some existing java class objects utilized in automation of adaptation plan generator which need to be introduced:

**Application Instances (I)** This object is used to represent any particular software component prototype, which is used for its different state.

**Application State (S)** To show different state of application instances created. There are 8 different states available in this object, which are NOT_SET, INIT, READY, ACTIVE, HOT, COLD, DEACTIVATED, ANY.

**Instance State** This object encapsulates both the objects mentioned above and links them together to identify the state of application instance.

**Control Unit Configuration** This object is used to represent available resources in the system. Configuration parameters which are required for each ECU in this failure management module SAPC are encapsulated in this object. Parameters such as ECU ID, its name, list of local failure events within the same ECU and all the application instances with their state (I, S).

**Adaptation Plan (AP)** This class is the important one which has all the necessary plans for mode transition. Each adaptation plan has unique id and name to identify. Also, this class encapsulates the information regarding current schedule table and new adapting schedule table. Similarly, tree map of all application instances with their state for active mode and also for the target mode, so that transition can be made accordingly.

**Configuration Manager** A complete manager which is responsible for all the actions necessary for creating an actual adaptation plan by using all the classes explained before. Configuration manager has all object creation functionalities e.g., creation of schedule table, control unit configuration, application instance with corresponding ECU and state. Also, it includes a list of all the adaptation plans which are configured for each transition. At the end, this object is used to generate the source files for SAPC module as “.h” and “.c” files, which later will be compiled and flashed into the hardware directly.

### 2.3 AUTOSAR Tooling

Several papers are released on AUTOSAR tooling to fill the tooling gap in the field of model driven application development. In these papers, [30] a method to generate configuration GUI from AUTOSAR description by constructing annotated Java class models is introduced. Here, a simplified way to configure control unit configuration parameter was developed. Again with same base, another paper [32]
2 State of the Art

implements a tool which provides traceability of safety-critical arguments. The approach of this profile enables the possibility of tracing automatic communication relevant properties [33], but no actual configurator was introduced. So, there has never been a research for configuring control unit parameters of AUTOSAR automatically, since it was unnecessary, but reconfiguration of system during runtime for each scheduled operational mode [37, 38] demands such automatic configurator for AUTOSAR modules.
3 Technology

This section of the thesis covers all the technology tools and the standards. These standards are referred and surveyed in order to build the concept behind the implementation of this master thesis. Each tool and technology is understood from the point of present research work and the utilization of the same. The explanation to the usage of each is covered under their respective topics. Since this thesis is based on the AUTOSAR framework, most of the norms used are related to utilized AUTOSAR 4.2. For this reason, background knowledge of the AUTOSAR architecture and methodology is important to be understood, which is explained in the subsection with all the standards referred to in this thesis.

3.1 AUTOSAR

“AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry ”[1]. AUTOSAR was founded in the year 2003, with the goal to avoid regular redevelopment of equal or similar software components with the help of standardization in the software development for control units, which tackle the increasing complexity. The system architecture includes a uniform and manufacturer-independent system platform, which shall provide a reduced development effort and, for each manufacturer, a quality intensification. So, the main focus of AUTOSAR is on scalability of function, transferability, re-usability. The architecture of AUTOSAR is introduced and explained briefly in next subsection.

3.1.1 Layered Software Architecture

An AUTOSAR control unit is divided into several layers as shown in Figure 3.1: The application layer, the runtime environment and basic software. Again, The basic software consists of three horizontal layers: the service layer, the ECU abstraction layer and the micro-controller abstraction layer.

The Application Layer

The application layer contains the implemented customer-functionality. In this layer, the layer-model differs to a component-model. The application layer consists of software components, which are defined by two characteristics.
First, the description of a software component is an XML file including details such as communication connections, interfaces, the body and its internal structure, time response and required resources of hardware. Second is the implementation which can be provided as source code or object code.

A software component communicates with its environment via ports. Two software components can only communicate with each other if their ports have compatible interfaces. Also, software components are differentiated between atomic and compositional. A composition is simply a logic aggregation of different components, which can be distributed over several control units. The atomic software components are again divided into application and sensor/actuator type software component. Furthermore, there are two different models for communication between software components. Sender-Receiver model broadcasts the information to one or all receivers, but receiver decides to accept or to discard the information. Whereas, in the Client-Server model, a client requests service from the server; so the communication is initiated by the client. Normally, Client-Server model is used to communicate with the hardware.

The Runtime Environment

The Runtime Environment (RTE) is a middle layer between the application layer and the basic software. The RTE consists of generated C-code, which realizes inter or intra software components communication and also between software components and basic software. From the point of view of a software component, it does not matter if it is intra-ECU or inter-ECU communication. With the inter-ECU communication,
the information is transmitted to the target device by the communication-stack of
the basic software and an available bus system. So, the idea is that all software com-
ponents from all control units can communicate as if they were connected in one big
control unit. At the intra-ECU communication, software components placed in one
control unit share information among themselves. This information exchange must
not be direct - it has to be realized by the RTE. So the outsourcing of one of the
communicating software components into another control unit does not mean any
expenditure. Only timings have to be considered at this reorganization because of
differences in the runtime behavior of the software which can be expected because of
transmission delays. As shown in Figure 3.3, RTE is also one of the layered modules
in AUTOSAR which needs to be configured while scheduling the system.

![Figure 3.2: VFB To RTE Mapping](image)

AUTOSAR Run-Time Environment is basically an implementation of Virtual
Functional Bus (VFB) concept together with OS and other basic software modules
[11]. The VFB is the sum of all communication mechanisms (and interfaces to the
basic software) provided by AUTOSAR on an abstract (technology independent)
level. When the connections for a concrete system are defined, the VFB allows a
virtual integration in an early development phase. RTE implements this VFB func-
tionality on a specific ECU. Such VFB to RTE mapping is shown in Figure 3.2. This
illustrates that all SWCs which were connected by a single virtual function bus are now mapped on to three different ECUs. The RTE provides the infrastructure services that enable communication to occur between AUTOSAR software-components as well as acting as the means by which AUTOSAR software-components access basic software modules including the OS and communication service. Basically, RTE is responsible for communication between SWCs and also scheduling of SWCs. Also, runnables get access to hardware-sourced data through the AUTOSAR RTE. The RTE provides the runtime interface between runnables and the basic software modules.

The RTE generator is one of the tools which actually realize the VFB for an ECU based on ECU configuration description file. The RTE generator also creates a link between SWCs to the OS, Basic software modules and also manages Intra SWCs communication. The RTE generation goes through two phases. First, during the RTE Contract Phase, using AUTOSAR interface definition and component information, an application header file is created which defines the contract between component and RTE. Next is the RTE Generation Phase. This generates one RTE for each ECU using SWC information (Runnable Entities, RTE Events), their placement to ECU and communication connections. Therefore, for configuration of each ECU, the RTE also needs to be configured. So, before configuring RTE the configuration tools need to gather all the information required to form an operating RTE. This information includes SWC instances and their communication relationships, the Runnable Entities and the involved RTE-Events. This data can be extracted from ECU Configuration Description or ECU Configuration Values, which also might provide references to further descriptions like the software-component description or the System Configuration description. Similarly, RTE configuration tool can use the information to create and check the mapping of RTE Event to OS tasks.

The Basic Software

The basic software is an abstraction layer of the hardware and shall prevent the direct access of the application layer onto the hardware. With the help of the basic software, reusability of the software components in the application layer, which realize the functionality, is possible. The basic software consists of several software modules, which have to be configured depending on the requirements of the application layer and on the hardware preconditions.

As per Figure 3.1, there are three horizontal layers of basic software, starting from bottom layer which are Microcontroller Abstraction Layer (MCAL), ECU Abstraction Layer and Service Layer. MCAL is the lowest hardware level, which is hardware dependable and has access to microcontroller drivers, memory, communication and I/O drivers. The goal of MCAL is to make ECU Abstraction layer hardware independent. Above MCAL there is ECU abstraction layer, which provide APIs to external peripheral components of ECU. This layer is mostly hardware independent.
3 Technology

Figure 3.3: AUTOSAR Layered Modules [2]

The upper most one is service layer, which provides background services like memory administration, network services or bus communication. The services layer is independent of the hardware.

In Figure 3.3, the module view of basic software is shown. In the figure, all the possible modules are displayed. According to the requirements from the application layer, only required modules are used and flashed to the control unit. Each necessary module is configured according to the requirement of application layer. Naturally, not all the modules are relevant to this thesis since it is focused on the scheduling related problem. Operating System is one of the modules from service layer of AUTOSAR which is relevant to scheduling properties of system. Important aspects of the Operating system are mentioned in the next section.

OS

According to [10], “Industry standard OSEK OS (ISO 17356-3) is used as the basis for AUTOSAR OS. The OSEK/VDX Operating System is widely used in the automotive industry and has been proven useful in all classes of ECUs found in modern vehicles”. OSEK OS is an event-triggered operating system. This provides high flexibility in the design and maintenance of AUTOSAR-based systems. Therefore, core functionality of the AUTOSAR OS is based on the OSEK OS. Fixed priority-based scheduling, facilities for handling interrupts, only interrupts with higher priority than tasks, protection against incorrect use of OS services and startup/shutdown
interface through functions are some highlighting features of OSEK OS to support AUTOSAR.

![OSEK Task State Model](image)

**Figure 3.4: OSEK Task State Model**

**OS Task** OS task is a container to represent OSEK Task [10]. During configuration of an AUTOSAR system, runnables of SWCs are mapped to the tasks (explained in 4.2) that are scheduled by the operating system. All runnables in a task share the same protection boundary. OS task state transition model is shown in Figure 3.4 based on [10]. On contrary within OS task, the execution of runnable follows pattern shown in Figure 4.6, where these set of events (e) are invoked on expiry points of the schedule table. Therefore, there is always a possibility of missing an event if runnables with overlapping execution time are mapped together in the same OS task. This topic is further explored in detail during runnable-to-task mapping algorithm.

**OS Events** OS events (e) are set of events which are activated on an expiry point in schedule table. It is possible to activate a task and set (one or more unique) events for the same task at the same expiry point [10]. An event setting can be done at each expiry point by referencing it to the created Event object. Also, tasks can be referenced to the OS events, so that invocation of events also starts running the task.
The schedule table implements statically defined task activation mechanism using OSEK counter and a series of auto started alarms, which consequently solves synchronization issue by defining set of expiry points statically. On each expiry point, one or more action must be processed, where action can be activation of tasks or invocation of an event as shown in Figure 3.6. Also, each expiry point has an offset from the start of the schedule table. The schedule table has a table period duration which is measured in ticks.

At runtime, the Operating System module will iterate over the schedule table, processing each expiry point in turn. The iteration is driven by an OSEK counter. It therefore follows that the properties of the counter have an impact on what is possible to configure on the schedule table.

Figure 3.5 exhibits different states of the schedule table and the transition between them. The schedule table does not execute during SCHEDULETABLE_STOPPED state. OS processes expiry points on the schedule table only in SCHEDULETABLE_RUNNING state. On the call of switch, it enters into SCHEDULETABLE_NEXT state and waits until the present executing table is finished.

According to [10], "An AUTOSAR OS must be capable of supporting a collection of OS objects (tasks, interrupts, alarms, hooks etc.) that form a cohesive functional unit. This collection of objects is termed an OS-Application. All objects which belong to the same OS-Application have access to each other. Access
means to allow using these objects within API services. Access by other applications can be granted separately". Basically, an OS application is another container which holds every OS object together. One ECU can have multiple OS applications. This is also one of the objects necessary to consider while configuring OS module automatically.

3.1.2 Design-Tool Work Flow (Methodology)

There is a general technical approach to develop a system, which is called as “AUTOSAR Methodology”. AUTOSAR Methodology is a work-product flow, which defines the dependencies of activities on work results.

Referencing Figure 3.7, the first step is to specify initial data for the design or the architecture of the system being developed. That means to select the target hardware ECUs and the software components SWCs. That means to map the components by regarding timings and resources onto control devices. The result is a “System Description” as AUTOSAR XML file, which contains the complete system information as bus-mapping, topology and mapping of software components on control units. The following steps are processed for each control unit for its own, thus no longer for the complete system. There is a process called Flattening, which is used to generate ECU specific information and this specific information is stored in “ECU Extract” of System Description. ECU Extract is similar to the System Extract of System Description, but only contains the atomic SWCs in a flat perspective. The following activity “Configuring ECU” feeds all the necessary information such as task scheduling, parameters for basic software modules and allocation of runnables to tasks on the control unit. This information is stored in an “ECU Configuration Description”. In the last step “Generate Executable Code”, a flashable file,
which contains the basic software, the RTE code and the software components of the application layer, is generated on the basis of the previously generated “ECU Configuration Description”. Being one of the main goals of AUTOSAR, simplifying integration of application components from the OEM and service providers, process of application development is independent of the above methodology steps. All the interfaces-related information of SWCs are described in the SWC Descriptions file (AUTOSAR XML file). Based on this description SWCs can be tested and implemented independently. As a result integration becomes easier. An analysis to understand the data structure of this configuration methodology is done in the next section.

3.1.3 ECU Module Definitions

As studied before, AUTOSAR uses XML schema for data exchange throughout the configuration methodology. This XML schema encapsulates the entire necessary data format during the process. Since, this research focuses on ECU configuration, there are only three relevant AUTOSAR schema elements: Module-Def, Module-Configuration and ECU-Configuration. Data structures of these elements are discussed in detail with simplified UML diagram [30] based on AUTOSAR standard [9] for easy understanding.

**Module-Def**  This element provides specification for configuration components. According to the AUTOSAR XML, schema Module-Def has a number of elements
and all of them have subclasses identifiable with properties such as Short-Name (name of an element) and Lower/Upper-Multiplicity (frequency at which it can occur in ECU configuration). Figure 3.8(a) visualizes UML diagram of these elements. This is the top-level component used to define the module whether the module is RTE or BSW module within which multiple sub containers exist.

Container-Def is used to define the specification of these sub containers. Container-Def has two types of definitions. First, Param-Conf-Container-Def, which defines configuration container containing other containers, parameter definitions and references. Second, Choice-Container-Def, which gives a choice between multiple Param-Conf-Container-Def.

Config-Parameter has multiple ECU configuration parameter types e.g. Float, Enumeration, Integer etc. as shown in the diagram in Figure 3.8(b). As shown in Figure 3.8(c), Config-Reference is another element with reference to the container within ECU configuration values or element belongs system model.

**Module-Configuration** This container holds final configuration values of corresponding software module which contains multiple components such as Parameter-Values (type values like Boolean-Value, Integer-Value), Reference-Values (reference to the path of container) and Sub-Containers (iteratively containing same elements Parameter-Values, Reference-Values, Sub-Containers).

**ECU Configuration** It is a collection of all Module-Configuration values which are configured for specific ECU. This also has reference to multiple modules which refer to multiple Module-Def. As shown in the Figure 3.9 from ARCTIC Studio
Technology

Figure 3.9: ECU Configuration File - Tool-View

(ECU configuration tool), where “EcuConfigControlUnit1.arxml” is ECU configuration file, which is a collection of several Module-Configuration values (RTE and OS is also listed in this).

3.1.4 Timing Extensions

Automotive systems are real-time embedded systems that need to justify their timing requirements for safe operation. In order to support developing such a system with real-time constraints and to perform timing analysis or validation of the developed system, timing extensions are introduced in the AUTOSAR [13]. These extensions allow the developer to configure the timing behavior of the communication stack and also make it possible to mention the execution time of executable entities.

AUTOSAR Timing Extensions can provide timing information using the Timing description. Timing constraints can be applied either on Timing Description Events, Timing Description Event Chains, or also on an ordered list of Executable Entities [13]. These timing extensions are encapsulated in a System Timing AUTOSAR object. There can be multiple system timing elements with different timing constraints for the same executable entity. In Timing Description Events, timing
constraint classifies a single event or a group of events with a temporal restriction, for example a period, latency or a time interval considered as synchronous. Also the direction has to be considered, which means in the semantics of the constraint it matters whether an event source (forward semantics) or an event sink (backward semantics) is considered. There are different types of timing constraints, but the scope of this thesis only needs the knowledge about Event Triggering Constraints.

![Figure 3.10: Concrete Pattern Event Triggering Periodic Repetition][13]

**Event Triggering Constraints** The element `EventTriggeringConstraint` is used to specify the particular occurrences of a given timing description event. AUTOSAR offers five types of event triggering as depicted in [13]. In our project, `ConcretePatternEventTriggering` is used. This element is used to specify the characteristics of a timing description event which occurs as a concrete pattern. The Concrete Pattern Event Triggering is characterized by the parameters Pattern Length, Offset, Pattern Period and Pattern Jitter.

Figure 3.10 characterizes these parameters very accurately. Pattern period from one of the pattern is used considering that it will be the same for the entire event pattern, which further can be used to configure the period of operating system schedule table. Schedule table period will be the global time scale for runnable execution during the runtime of the system. Also, while configuring this pattern, there are some constraints applied to `ConcretePatternEventTriggering` and shall be considered when using this event triggering constraint as shown below.

\[ 0 \leq \max(\text{Offsets}) \leq \text{Pattern Length} \]
### Table 3.1: Semantics of cseCodeType [13]

<table>
<thead>
<tr>
<th>cseCode</th>
<th>Time Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1µsec</td>
</tr>
<tr>
<td>1</td>
<td>10µsec</td>
</tr>
<tr>
<td>2</td>
<td>100µsec</td>
</tr>
<tr>
<td>3</td>
<td>1 msec</td>
</tr>
<tr>
<td>4</td>
<td>10 msec</td>
</tr>
<tr>
<td>5</td>
<td>100 msec</td>
</tr>
<tr>
<td>6</td>
<td>1 sec</td>
</tr>
<tr>
<td>7</td>
<td>10 sec</td>
</tr>
<tr>
<td>8</td>
<td>100 sec</td>
</tr>
<tr>
<td>9</td>
<td>1 hour</td>
</tr>
<tr>
<td>10</td>
<td>10 day</td>
</tr>
</tbody>
</table>

Pattern Length + Pattern Jitter < Pattern Period

This **ConcretePatternEventTriggering** element has an essential property known as “Event” reference. This references to the object **TDEventsSwcInternalBehavior** which has reference to the required runnable. Also, it carries trace to the corresponding software component in system model using **ComponentInCompositionInstance Ref**. **TDEventSwcInternalBehavior** is used to specify events, namely the activation, start, termination of runnable entities, as well as variable accesses, which are observable in the Software Component view. Therefore, there are three different concrete event patterns for activation, start and end, where each refers to the corresponding **TDEventSwcInternalBehavior**.

To specify the exact measure or value of time, it is very important to mention a clear explanation of the timing properties such as unit seconds or angular degrees. **ConcretePatternEventTriggering** pattern includes an AUTOSAR model element **MultiDimensionalTime**, which defines timing properties with the use of ASAM CSE codetypes. This element has two attributes **cseCode** and **cseCodeFactor**. CSE-code defines the base of time and a scaling factor is mentioned in CSE code factor to determine the final value. Such semantics of the CSE-code is given in Table 3.1. If, for example, **cseCode** is 0 and **cseCodeFactor** is 100 then according to Table 3.1 it determines value as 100 usec; similarly if **cseCode** is 100 and **cseCodeFactor** is 300 then it represents a value of 300 angular degrees.

### 3.2 Technical Background

**Eclipse**

It is an open source software development platform which is composed of multiple framework and tools. This platform contributes to the several sectors
including the information technology sector. This is utilized to develop new feature and functions by using the Eclipse plug-ins.

**Eclipse Plug-Ins** An application developed in eclipse has multiple software packages as a component. These components are called “plug-ins”. Each plug-in developed by a programmer can create menu or toolbar entries. This kind of approach is largely acknowledged in the field of software tooling. A tool such as Eclipse IDE supports development of these application components. Also, a concept of runtime Eclipse IDE helps to test a developed set of plug-ins by creating another instance of Eclipse IDE. One workspace can have multiple plug-in projects depending on each other, where each project provides different features to the developing tool. Eclipse IDE Neon.1a Release (4.6.1) version is used for the development of plug-ins in this thesis.

**dSpace SystemDesk** SystemDesk from the company dSpace is a software architecture tool to support the development of distributed automotive electric-/ electronic systems. SystemDesk allows a model-based development of control unit-software and displays their components and communication graphically. Beside the integration of control unit, RTE code and AUTOSAR-compliant interfaces for software components can be generated by SystemDesk. In the context of this thesis, SystemDesk is used to extract ECU extract file and to read or write back modified system model from developed tool feature. SystemDesk version 4.6 is used during this thesis.

**ARCCORE-Arctic Studio** “The Arctic Studio tool chain provides a complete embedded software development environment for automotive embedded software based on the open industry-leading standard Autosar ”[7]. ARCCORE is a vendor who provides AUTOSAR development products and Arctic Studio is one them. It is a supporting tool chain for automotive developing stage including platform development. During platform development several embedded software modules of AUTOSAR basic software stack need to be configured, which can be done by Arctic Studio. During this thesis work, configured parameters in developed features for system service and interface module are checked over Arctic Studio version 15.0.1 tool.

**Java Xtend** “Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects ”[18]. Xtend is used for transforming models into other models, for modifying existing models before they are fed into a code generator, as well as for implementing simple helper functions that are used as part of code generation templates [26].
WinIDEA  This is the flashing tool used for mapping generated source files for different modules into the hardware. “winIDEA” is IDE product from iSystem for software development of embedded system, which helps tracing and debugging source code.
4 Concept

This chapter gives an overview of the current design flow and explains the part at which work of this thesis is embedded in available design work flow. Also, introduces fundamental concepts necessary to understand, on which implementation of this master thesis is based on.

4.1 Overview of Design Flow Integration

Safe adaptation is modelled on different level of abstraction during the process, which is based on modelling language such as Unified Modelling Language (UML). Eclipse modeling framework (EMF) is also a model driven application development framework, which makes it possible to develop a tool with the support of ARTOP to edit or to model a system as per the requirements. AUTOSAR is preferred for the design flow since it describes the higher level abstraction in Electric or Electronic system. This design work flow has five stages of tooling chain: Application design, Architecture, Planning, Configuration and SAPC as shown in figure 4.1. Where, The work of this thesis lies within the last two stages of the design flow.

![Design Work Flow](image)

Figure 4.1: Design Work Flow [19]

4.1.1 Application Design (System Modeling)

In this stage of the design process, individual function components are designed independent of the involved hardware. An application in AUTOSAR consists of
interconnected AUTOSAR Software components. These components are modeled and configured with timing requirements and further exported as AUTOSAR system description. Failure Mode and effects analysis (FMEA) and Failure Tree Analysis (FTA) is used to validate and verify developed Software components. FMEA is a bottom-up technique used to identify, prioritize, and eliminate potential failures from the system, design or process before they reach the customer. FTA is a top-down failure analysis used for discovering the root causes of failures or potential failures. Only introduction is given since it is not in the scope of the thesis work.

4.1.2 Architecture (Operational Modes Environment)
Graceful degradation requirements are configured in this stage of the design flow. Even though this part is not in the thesis work, some initial work is done on this stage. To configure and modify the system model as per the requirements of different system mode and timing details of SWCs, an Eclipse based editor is created with the support of ARTOP using Eclipse plug-in projects. This editor has multiple windows for different purposes of the project.

One of the aims of Architecture is to configure different modes with the composition of SWCs and available resources using AUTOSAR model objects such as ModeDeclaration, ModeDeclarationGroup and Collection. Each object with unique properties is utilized to declare and define system modes. There are multiple views for different objective, one of which is to configure system modes. As shown in figure 4.2, there is an eclipse-based view called SystemModeView which allows the user to define discrete modes with different constraints and properties. As per figure 4.2, three modes are configured, where each mode has available ECUs under resources, SWC prototypes under Tasks and some Constraints for resources to handle functionalities mutual exclusion. In this way, all failure environments are configured in the form of system modes. Later, this group of system modes that are accumulated in ModeDeclarationGroup type object called SystemModes is utilized to create adaptation plans.

![Figure 4.2: System Mode View](image)
Timing view is another Eclipsed-based view, which graphically represents the interconnection between SWCs and their communication deadline value. This view is also used to edit the timing description of communication links. As shown in figure 4.3, each block represent atomic SWC prototype, where red colored block is internal Atomic SWC of selected composition SWC prototype. Similarly, blue colored blocks are external SWC prototypes which are communicating to the selected prototype. This makes communication easy to visualize and edit according to the requirements.

Result view is similar to the previous one on development perspective. This view is more useful to evaluate the planned system model. This view gives a graphical representation of SWCs runnable execution model for each mode after the system is scheduled by the planning stage, which is the next stage after Architecture. As shown in figure 4.4, execution model for selected system mode “FailureECU1”, where it shows execution of each SWC runnables belongs to the mode and their pre-emption to each other, for e.g., IoHwAb(SteeringProto1) pre-empted SteeringWheelMessageRelayRunnable(SteeringProto1), is shown as dark red strip during execution of SteeringWheelMessageRelayRunnable(SteeringProto1). This view is a vital part of evaluation setup, as it will be used as a reference graph to cross-check required runtime of runnable.

4.1.3 Planning (Scheduling)

After system model is configured with system modes and their requirements in architecture, it goes through planning stage which infuses scheduling information into the same system model. “This additional scheduling information poses as an extension of the system’s interface description to guarantee the required real-time behavior when implemented correctly on each control unit, thus providing the basis for compositional system integration”[37]. Mixed-Integer Linear Programmes have been acknowledged in real-time system integration [37]. MILP representation of developed system model from Architecture stage is formulated. From this formulation valid schedules are defined for required set of known compositions for each
Due to comprehensive form of MILP, planning stage forms the most distinctive and complicated equation with number of variables to get the schedules for system modes. Again, this is outside the scope of this thesis work.

## 4.1.4 Configuration

The output of planning stage generates AUTOSAR exchange description file with scheduling information from system perspective. In configuration, AUTOSAR parameters of basic software and RTE have to be configured from ECU point of view. This stage also aims at configuration of scheduling-related AUTOSAR ECU-modules based on a centrally synthesized system schedule, but configuring such scheduling modules in orthodox manual way is inefficient, difficult and potential human errors considering amount of system modes. Hence, research on this area led to development of a process to automate the configuration of scheduling related AUTOSAR ECU-modules from system schedule during work of this thesis. These scheduling relevant modules are nothing but OS and RTE to achieve real-time requirement. An algorithm to generate configuration of OS and RTE module is introduced in chapter 5. There is a concept while configuring such modules in AUTOSAR to allocate real-time task such as runnable to OS task. A concept which can counter problems
of different constraint for a such mapping is discussed and proposed in detail during section 4.2.

4.1.5 SAPC

This is the last stage of the design process which is already explained in detail during section 2.2.2. This module provides the service of monitoring and runtime re-configuration. As mentioned in 2.2.2, SAPC still lacks in generic automatic adaptation plans generation. To have such a generic automated service which will analyze the developed system model and configure adaptation plans from formalized information needs a proper search algorithm to go through each hierarchy level of system modes shown in 4.10. This search algorithm is introduced and explained in section 4.3.

The next part of the chapter explains the concept behind the implementation. This thesis works on providing automated generation service to configuration stage of design flow. Block diagram of such an automated service is shown in figure 4.5. Automatic configurator takes an input from ECU extract, planned system model and ECU configuration file. Further, it goes through different stages (more detail in 5) to handle functional criticality and their preemption on real-time execution. Initial two stages are for runnable partition necessary for runnables to task mapping which is explained with detail in next section 4.2. Also, search algorithm used for navigation through system modes tree to generate adaptation plans is introduced in section 4.3.

Figure 4.5: Block Diagram for Automated Configuration
4.2 Scheduling of Mixed Critical Runnable

This section covers aspects of real-time system programming. Normally, system developers architect a system in such a way that it contains group of runnables with some real-time constraints. So, each piece of functionality that has to be executed during runtime of the software component is wrapped around runnable.

Runnable that represent software component can be activated by RTE and then executed by the task of the operating system. During ECU configuration, there is a concept called runnable to operating system task mapping, which is later used by RTE to define scheduling and execution order according to the requirements. This mapping is performed based on ECU Configuration description file. Once runnables are mapped on to the task, such set of tasks are executed during runtime on OS schedule table. A task set which is dispatched to the processor can either be time-triggered or event-triggered, where each task can have priorities such that higher priority but in the scope of this project we are using event triggering task which is also known as extended tasks. The OSEK task execution model with its state machine diagram is given in the figure 3.4.

![Figure 4.6: OS Task Runnable Execution](image)

If runnable with mixed criticality is grouped together to the same task then there is a possibility of lower critical application affecting the execution of a higher critical application. The OS task has a fixed amount of memory available for all the runnables present in the task. The quality of standard of developing safety critical application is much higher than lower critical application. Due to such quality difference it is not safe to group such runnables together in the OS task sharing the same memory. Data corruption caused by lower critical function might lead to unexpected behavior in critical application. So, it will be necessary to consider ASIL level of runnable during runnable to task mapping in RTE configuration.
All the runnables belonging to the same task are executed sequentially on the invocation of the corresponding events, since used tasks are extended type [10]. If two runnables exist in the same task, the one which has start and end execution period within execution period of the other runnable then those event will be missed by the task. Therefore, next events can be read only after the execution of the first runnable because of the manner in which task executes its jobs or runnable in this case. For example in the figure 4.6, there are two runnables in the task. When the task starts its execution, it gets an event E1 and Runnable1 starts running, but during live period of Runnable1, two events get invoked for Runnable2 (E2, E3). These events for Runnable2 get overlooked because Runnable1 is still alive. Now, after execution of Runnable1, there is another event E4 for Runnable2, which will be accepted and now Runnable2 will start its execution. So because of this sequential execution of runnables in the same task, partitioning runnable together which preempt each other is not possible, which is also another consideration while grouping runnables into one task. A proper method which will consider such a preemption condition before allocating runnable was required.

Conflict graph and graph coloring algorithm was invented in the area of hardware software co-design, where it is used while code generation to optimize the number of physical registers on the hardware. Therefore, it is known as a register conflict graph where two register conflicts with each other if lifetimes of them overlap within single basic block (BB) of code. As per figure 4.7, if we assume Ri (where i = 1, 2, 3, 4) are registers from code generation context then link between register R1 and R3 represent lifetime overlap. This same algorithm is now utilized in this thesis implementation where a graph represents the conflict or preemption between two runnables. Now, the conflict graph describing runnable execution time overlap causing for preemption is called as Runnable Conflict Graph (RCG).
Therefore, the modified definition of RCG will be as follow: A runnable conflict graph \( G=(R,E) \) is an undirected graph, where the nodes \( R \) represent the variables (symbolic runnable) and the edges \( E \) represent the conflicts between the runnable. An edge between \( R_i \) and \( R_j \), indicates that \( R_i \) and \( R_j \) preempt each other. Therefore, \( R_i \) and \( R_j \) cannot share the same runnable group.

Now, once the graph representing conflict between two runnables is found, there was a need of phenomenon which can group two non-conflicting runnables together. From a research on this topic resulted in finding two related algorithm which are Graph Coloring and Welsh Powell algorithms from the same field of conflict graph. These algorithms are discussed in detail in next subsection of this chapter.

**Graph coloring algorithm**  Basically, graph coloring algorithm color the nodes of \( G=(R, E) \) with k colors in a way that no two adjacent nodes get the same color (NP-complete problem), as shown in the figure 4.7 considering above example. Then, there is question whether a graph \( G=(R, E) \) can be colored with k colors, so the solution to this problem involves three steps:

I find a node \( R_i \) in \( G \) with \( \text{deg}(R_i) < k \)

II create \( G' = (R', E') \) by removing \( R_i \) and all its edges

III if \( R' = \text{empty} \) then k-colouring is possible else if all nodes \( R_i \) in \( G' \) have a \( \text{deg}(R_i) \geq k \) then k-colouring not possible else \( G = G' \), goto 1

![Figure 4.8: Greedy Graph Coloring](image)

The smallest integer \( k \) for which \( G \) is k-colorable is called the chromatic number of \( G \). Therefore, a graph whose chromatic color is \( k \) is called a k-chromatic graph. So, for coloring the graph this implementation is using basic greedy algorithm to go through each node. Following steps are involved in greedy algorithm:

I Color first vertex with the first color
II Consider the currently picked vertex v and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on the vertices adjacent to v, assign a new color to it.

III Do step II for remaining vertices until all the vertices are colored.

So, if the graph example from previous section is considered figure 4.7, then following the same three steps of this algorithm will give you a result of colored as shown in the figure 4.8.

From analysis of greedy algorithm, it does not always use minimum number of colors. Also, the number of colors used sometimes depends on the order in which vertices are processed. For example, consider figure 4.2 with two graphs. Note that in figure 4.9(b), vertices R4 and R5 are swapped. If we consider the vertices R1, R2, R3, R4, R5 in left graph, we can color the graph using 3 colors. But if we consider the vertices R1, R2, R3, R4, R5 in right graph, we need 4 colors. Hence, the introduction of Welsh Powell Algorithm which makes the order of nodes in manner to use the possible minimum number of colors, steps involved in this algorithm is as follows:

I Find the degree of each vertex

II List the vertices in order of descending valence i.e. degree($R_i$) ≥ degree($R_{i+1}$)

III Color the first vertex in the list

IV Go down the sorted list and color every vertex not connected to the colored vertices above the same color then cross out all colored vertices in the list

Figure 4.9: Greedy Analysis
4 Concept

But this thesis test bench class is based on greedy algorithm, since it is an initial prototype. For future research on this topic, Welsh Powell Algorithm will be the best option for graph coloring.

To solve this problem of graph coloring, this thesis is using the most obvious solution, which is Backtracking. This includes the following steps:

I Number the solution variables \([R_1, R_2...R_{n-1}]\) Back: 1

II Number the possible values for each variable \([C_0, C_1...C_{k-1}]\)

III Start by assigning \(C_0\) to each \(R_i\)

IV If we have an acceptable solution, stop

V If the current solution is not acceptable, let \(i = 0\) and increment till \(i \leq n-1\)

VI If \(i > n-1\), stop and signal that no solution is possible

VII Let \(j\) be the index such that \(R_i = C_j\). If \(j < k-1\), assign \(C_{j+1}\) to \(R_i\) and go back to step

VIII But if \(j \geq k-1\), assign \(C_0\) to \(R_i\), decrement \(i\), and go back to step VI

Although this approach will find a solution eventually (if one exists), it isn’t speedy. Backtracking over \(n\) variables, each of which can take on \(k\) possible values, is \(O(kn)\). For graph coloring, we will have one variable for each node in the graph. Each variable will take on any of the available colors.

In context of this thesis from the output of stage 2 referencing figure 4.8, each color can be interpreted as OS tasks, in which same colored vertices are runnables that can be grouped together. So, grouping runnables belonging to the same OS task, not only do they preempt each other but also belong to different ASIL safety level. This collection of optimum possible runnable grouping is the final stage 3 output of this heuristics. Therefore, we have a set of tasks \(\{T_0, T_1...T_{k-1}\}\) rather than a set of colors, whose cardinality is same as that of color set \(k\). Now, these sets of tasks are nothing but a set of strings with different short names since creation of actual OS task is part of OS configuration algorithm. Nonetheless, it is easier to consider them as OS task.

An algorithm is developed for implementation in 5.2 which carries out runnable to task mapping considering criticality of functions in the form of ASIL level and their preemption condition.
4.3 Algorithm For Hierarchical Reconfiguration Plans

SAPC is a module which handles the transition of a system from one mode to another, such as system modes \( M = \{ m_0, m_1, ..., m_y \} \), where \( y \) is the cardinality of modes) represent the health state of a system. For example, if the system is in a normal mode then all the resources (ECUs) are considered to be available while in a failure mode, one or more ECUs may not be available. So, failure of any ECU might force a system to transit from one mode to another. These changeovers are carried out by having deterministic adaptation strategy for every possible transition of system.

![Generic Adaptation Tree](image)

Figure 4.10: Generic Adaptation Tree

Previously, in this module, all the adaptation plans were configured manually and statically. For example system model which was developed during previous project research. Considering one of the example models, this had total three ECUs. In view of generic mode adaptation flow (figure 4.10), this model also follows the same hierarchy of transition for three ECUs. So, in Normal mode all the ECUs are available, but if one or more of ECUs fails, knowing the failed ECU, adaptation plans are created. If in considered case, one of the ECUs is failed, the transition will be from Normal mode to FailMode1. Now, in FailMode1 mode, only two resources will be available and new states are configured for applications on available ECUs.
the only possible transition is from FailMode1 to either FailMode11 or FailMode12 considering fewer resources available than FailMode1. In such manner, all the possible transitions are configured manually until there is no possible transition, which makes this module appropriate for very specific system model. Therefore, SAPC demands a generic algorithm which can generate adaptation plans for any number of ECUs in the developed system model. Therefore, an approach toward generic and automatic adaptation plan creator for mode transition is exhibited during this thesis work. Implementation for this approach is discussed more in detail during section 5.4.

To generate the adaptation plans for given tree of modes with different hierarchical levels as shown in figure 4.10, it requires a proper search algorithm which can go through each mode from initial mode through all fail modes. There are several searching algorithm such as Depth-first search, Breadth-first search and Best-first search. Among these approaches best suitable search algorithm found for this thesis is Breadth-first search algorithm as shown in figure 4.11.

Figure 4.11: Breadth Search [4]

Breadth search algorithm travels through the tree by each hierarchical level, which is required for the adaptation plan generation. This search algorithm goes through nodes in order of a, b, c, d, e, f, g, h, i, j, k with flow from top to bottom which is the required flow of search for adaptation plan generator. This search algorithm is a vital part of generic adaptation plan generator implementation discussed in chapter 5.4.
5 Implementation

Implementation of this paper follows Eclipse plug-in development. Each class model instances or extensions are either written separately in Eclipse editor or inherited from existing project classes. With the support of interfacing framework such as ARTOP, it helps Eclipse in editing, modifying and rewriting the AUTOSAR XML exchange files. Algorithms for automatic configuration go through three stages of implementation, referencing the block diagram 4.5 given as follows:

I **Data Accumulation:** Collection of all relevant data from available description files such as ECU extract file, system model and also configuration file. All of these files are saved in the AUTSAR XML exchange format.

II **Partitioning For Task Allocation:** Finding the best possible grouping of runnables while considering their safety criticality levels acc. to ISO 26262 and preemption due to overlap of execution period. This stage involves sub steps to achieve the required result.

III **Automatic Module Generation:** Finally from available data and formed group of tasks, OS and RTE is configured.

Implementations of these stages are explained in detail in sections 5.2, 5.3.

5.1 Data Accumulation

Before configuring objects mentioned in 4.1.4, implementation algorithm needs to have all the relevant information. This stage of algorithms is to collect all relevant data from system model, ECU extract and configuration file.

At first, Flat Map, SWCs prototypes (P) and their runnables (Δ) are collected from ECU extract file (ECU\textsubscript{X}) which is just a AUTOSAR XML schema. ECU FlatMap is an AUTOSAR model object of type \textit{FlatMap} which links ECU abstract SWC to system level SWC and is extracted by using a function called \textit{getEcuFlat}. This object is important when the algorithm needs a link between referenced SWC in configured timing extensions from system perspective and ECU extract SWC prototype. Also, the respective module configuration values are extracted from the ECU configuration file to configure parameters of each module. This can be done by filtering configuration values with module definition. Hence, module definition is also another object necessary to extract from ECU configuration file. As studied
5 Implementation

in 3.1.3, each module definition includes all the parameter and reference definitions
within itself. The flow of data extraction within available files is given in Figure 5.1.

Initially ECU configuration file is created with a reference of the ECU extraction
file, which is the first link of the configuration file where the ECU extraction file
generated by any system model developing tool has information from the perspective
of a single ECU. This extract file holds information such as atomic SWC prototypes,
communication connectors and FlatMap. The second link between extract file and
system model is through FlatMap object. It shows which compositional SWC does
the given atomic SWC belong to at the system level. Both these links are illustrated
in figure 5.1, in which the first link is uni-directional whereas the second link is bi-
directional. In this way, extracting data from these files becomes easy with any
model developing tool.

![Figure 5.1: Data Extraction Links](image)

5.2 Partitioning For Task Allocation

A heuristic is developed for runnable to task mapping based on knowledge from 4.2,
which goes through three steps:

1. Create group of runnables belonging to each ASIL level, which counters the
problems of mixed criticality within the same task.
5 Implementation

2. Taking ASIL level runnable group from step 1 and for each group, get the activation offset of every runnable from system scheduling requirements and construct adjacency conflict matrix representing preemption possibilities between runnables using their offset values considering each mode scheduling description.

3. Finally, find the optimum possible grouping of runnable using step 1 and 2, which is compatible with every mode without any preemption for each ASIL level group.

Stage 1: Partitioning Runnables Considering ASIL level

This section explains in details about algorithm 1 whose objective is to categorize runnables according to their automotive safety integrity level referencing to ISO 26262. There are several levels for indicating criticality of application component. According to ISO 26262, there are five ASIL levels. \( X = \{QM, A, B, C, D\} \), where \( X \) is any ASIL level belonging to the set [12].

Let us assume a set of runnables \( \Delta = \{r_1, r_2, \ldots, r_p\} \) with cardinality \( p \) belong to ECU, since RTE configuration is done using ECU abstraction file from single ECU perspective. ASIL level of each prototype is traceable within its admin data, if the corresponding prototype is already evaluated for its criticality. In admin data, there is an element called SD element which holds information for ASIL value with \( gid \) string parameter as \( ASIL \) and \( value \) parameter holds actual value of level belonging to \( X \). Now, depending upon the ASIL value \( X \) of the prototype, runnable belonging to that prototype is assigned to the set \( ASIL_X \), where this set is subset of \( \Delta \) (\( ASIL_X \subseteq \Delta \)). From the algorithm, the return value is given to \( \Gamma \), which is a collection of all ASIL level runnables group such that \( ASIL_X \in \Gamma \). So, the initial requirement of this heuristic is that the set of prototypes \( P = \{p_1, p_2, \ldots, p_P\} \) belonging to ECU, which can be extracted from the abstraction file.

Stage 2: Runnable Offsets and Conflict Adjacency Table

From algorithm 1, collection of groups representing different ASIL levels is obtained, where each group is a set of runnables. This algorithm provides prerequisite information for the algorithm 5 to obtain final feasible group of runnables which can be assigned to the tasks. \( ASIL_X \) has a set of runnables whose criticality is identical; hence possibility of failure or data corruption due to mixed criticality within same task can be avoided. Also, having same development quality applications in the same task is not the optimum solution or does not solve the problem totally, but it should prevent the problem from occurring due to mixed criticality.

Now, the second consideration is for preemption between runnables considering their execution period. As studied in 4.2, the OS task execution model prevent from
Algorithm 1 Runnable Partitioning

1: **Data:** \( P \) prototype set
2: \( \Gamma \leftarrow \text{empty} \)
3: **for each** \( p \in P \) **do**
4:    **if** \( p\text{.adminData} \neq \text{null} \) **then**
5:       \( Sd \leftarrow \text{empty} \)
6:       \( Sd \leftarrow p\text{.adminData.Sd} \)
7:        **if** \( Sd\text{.value} \neq \text{null} \) **then**
8:           \( X \leftarrow Sd\text{.value} \)
9:           \( ASIL_X \leftarrow \text{empty} \)
10:          \( ASIL_X \leftarrow \text{getRunnable}(p) \)
11:         \( \Gamma \leftarrow ASIL_X \)
12: **else**
13:     \( ASIL_0 \leftarrow \text{empty} \)
14:     \( ASIL_0 \leftarrow \text{getRunnable}(p) \)
15:    \( \Gamma \leftarrow ASIL_0 \)
16: **end for**
17: **return** \( \Gamma \)

grouping runnable with overlapping execution period. Therefore, to check for this criterion, the algorithm needs to have information regarding runnable activation and end of execution to determine its execution period on the global time scale of the schedule table. The OS schedule table takes offsets for each runnable in terms of expiry points to invoke each event for execution of corresponding runnable, which asks for a function which extracts offset timing information for each runnable present. Therefore, this function (called `getRunnableOffsets` written in Java Xtend) needs to have prior knowledge about event patterns on which timing constraints are applied, which already exists in the system model AUTOSAR-XML.

As per the study from AUTOSAR timing extension introduction chapter 3.1.4, timing constraints are specified in concrete event patterns for each activity such as activation, start and termination of executable entity (runnable). In the other hand, this algorithm is only interested in concrete event patterns for activation and termination. This information will be enough to determine their execution period on the schedule table. `getRunnableOffsets` function extracts such offset information from concrete event patterns for each runnable in the form of an object (called “RunnableOffsets”), which includes runnable activation, start and end offset as double type time value in milliseconds. Once the offsets of all runnables are collected for the particular schedules of system mode, these offsets will be used to create a runnable conflict graph, which consequently, will be used to determine possible grouping of runnables that do not preempt each other using graph coloring method (referenced to 4.2).
In this algorithm, to represent such a conflict graph, a matrix is used in such a way that if runnable groups are of “n” size then the program creates matrix of “n x n”, where each co-ordinates has conflict values. So, the problem complexity becomes O(n^2), in which each runnable is checked across every other runnable present in the group, considering that a runnable does not conflict with itself.

Let’s assume that there is a list of 4 runnables Ri (where i= 1, 2, 3, 4) conflict among themselves as shown in figure 4.7a. So while developing this algorithm for 4 runnables, there will be a square matrix size of 4. Since this development needs a square matrix, a matrix of 4x4 is created, where rows and columns are the same group of runnables with index of i as shown in 4.7b. Each co-ordinate of the created matrix will have bit values where 0 means no conflict and 1 means conflict as shown in figure 4.7b. For this reason, there is a function called createAdjacencyMatrix which gives a matrix “AdjacencyMatrix” (AM) as an output representing conflict between runnable of ASIL_X. Hence, input to this function is ASIL_X and “RunnableOffset”. Algorithm 2 is to create such a matrix as shown in figure 4.7b. Initially, an empty square matrix is created with the default value as 0 in all co-ordinates with size of ASIL_X runnable group. Now, each runnable of this group is checked across every other member including itself. Therefore, there is a nested for loop to test such a complex problem. To decide the conflict between runnables within same group, there is another check function called isRunnablesPreempting. This function returns a boolean value on checking whether the passed argument test runnable (R_t) preempts the reference runnable (R_r) or not. The preemption condition checked in the function is as follows:

\[(R_t.\text{activation} < R_r.\text{activation}) \land (R_t.\text{end} > R_r.\text{end}) \lor (R_t.\text{activation} > R_r.\text{activation}) \land (R_t.\text{end} < R_r.\text{end})\]

Pre-emption Condition

**Algorithm 2** Conflict Matrix

1: **Data:** ASIL_X Runnable group, Offset for each \( R_t \in \Delta \), AM
2: \( n \leftarrow \text{sizeof}(\text{ASIL}_X) \)
3: for row = 0...n − 1 do
4:   for column = 0...n − 1 do
5:     \( R_t \leftarrow \text{ASIL}_X[\text{row}] \)
6:     \( R_r \leftarrow \text{ASIL}_X[\text{column}] \)
7:     if isRunnablesPreempting(\( R_t, R_r \)) then
8:       AM[\text{row}][\text{column}] \leftarrow 1
9:     end if
end for
10: return AM

52
If the above condition is true, then \textit{isRunnablesPreempting} function returns true, otherwise false using offsets values of runnables from “RunnableOffsets” object which is passed as an argument. On the occasion of preemption, the corresponding coordinates of the adjacency matrix (AM) are set to value 1, representing conflict. At the end, conflict matrix AM is returned.

The prerequisite data needed to be available are $\Gamma$ from stage 1, collection of scheduling requirements (SR) for each system operational mode $L$ (where $L$ length of $m$) for the implementation of stage 2 in algorithm 3. Here, $m$ is the cardinality of SR and $L$, which depends on the number of modes configured from architecture stage in the system model. Eventually, this means there is scheduling for each mode ($SR_L$). So, this scheduling requirement has all the offset information necessary for this algorithm in the form of concrete pattern event as discussed in previous chapter 3.1.4, which is collected in a java encapsulation object named “RunnableOffsets”. In this object, offset information is mapped to the runnable it belonging to as a value so that it can be easily extracted later on. Once this is achieved, using the offset information gathered and $ASIL_X$ runnable group, adjacency matrix (AM) is created using \textit{createAdjancencyMatrix} function from algorithm 2, which is a square matrix size of $ASIL_X$ group cardinality. This matrix is stored in local copy and then later updated into AM. The aim of this local copy and update function is to maintain or also to keep memory of conflicting runnables from previous modes. Therefore, AM is updated in such a way that it does not lose the conflict information throughout all the system modes.

\begin{algorithm}
\caption{Mode Specific Conflict Check}
\begin{algorithmic}
\State \textbf{Data:} $\Gamma$, $SR_L$
\For{$ASIL_X$ in $\Gamma$}
\State $n \leftarrow \text{sizeof}(ASIL_X)$
\State $AM[row][column] \leftarrow \text{empty}$
\For{$L$ in $SR$}
\State $Local[n][n] \leftarrow \text{empty}$
\State $RunnableOffset \leftarrow \text{getRunnableOffsets}(SR_L)$
\State $Local[row][column] \leftarrow \text{createAdjancencyMatrix}(ASIL_X, RunnableOffset)$
\State $update(AM) \leftarrow Local$
\EndFor
\EndFor
\State \textbf{return} AM
\end{algorithmic}
\end{algorithm}
5 Implementation

Stage 3: Final possible grouping of runnable

This last stage of algorithm focuses on finding the final possible grouping of runnables which will be mapped to the AUTOSAR OS tasks.

Algorithm 4 isSafe Function
1: Data: $R_t, T_i, i$ (Runnable index), AM, Tasks
2: $n \leftarrow \text{sizeof}(ASIL_X)$
3: AM[row][column] $\leftarrow$ empty
4: for $j = 0...n - 1$ do
5:   if (AM[i][j] = 1 && Tasks.exist($R_t, T_i$)) then
6:     return false
7: end for
8: return true

There is an acceptance condition at step VI of backtracking (refer to 4.2), which basically should check whether it is possible to assign color or in this case task to runnable. For this purpose there is a boolean function that takes a runnable and the task which needs to be tested across the available conflict matrix. Algorithm 4 exhibits the working of this boolean function known as isSafe.

Algorithm 5 Final grouping
1: Data: $ASIL_X, AM$
2: $n \leftarrow \text{sizeof}(ASIL_X)$
3: $T \leftarrow \{T_0, T_1...T_{k-1}\}$
4: $k \leftarrow \text{sizeof}(T)$
5: Tasks $\leftarrow$ empty
6: for $i = 0...n - 1$ do
7:   for $j = 0...k - 1$ do
8:     $R_t \leftarrow ASIL_X[i]$
9:     $T_i \leftarrow T[j]$
10:    if (isSafe($ASIL_X, T_i, i, AM$)) then
11:       Tasks $\leftarrow$ map($R_t, T_i$)
12:      $j = k - 1$
13: end for
14: end for
15: return Tasks
Our solution variables for this check are ASIL\(_x\) runnable groups which belong to \(\Gamma\). Also, another available data from second stage is adjacency conflict matrix \(AM\). Following backtracking steps 4.2, each runnable in ASIL\(_x\) will be checked across every task in this context. Hence the nested loop, where first loop for runnables and second for task set. \(R_t, T_t\) means test runnable and test task. As from algorithm 4, function \(isSafe\) will check whether \(R_t\) is possible to map to \(T_t\). If this function returns value true then \(R_t\) will be mapped to task \(T_t\) and stored in the global variable called \(Tasks\). Once the possible task is found, task iteration will be skipped and next runnable will be taken as \(R_t\). At the end, runnables will be grouped depending on the possibilities with different tasks as shown in figure 4.8 in greedy manner.

### 5.3 Automatic Module Generation

This is the last stage 4.5 of the configurator which generates scheduling related modules configured with required information from available data in the system model and acquired output from the previous algorithm 5.

#### 5.3.1 OS

During the process of module generation, object containers such as OS tasks, application, counter and schedule table with expiry points (refer to 3.1.1) are the essential elements for configuration as discussed in 3.1.1. As shown in the Figure 5.2(b), \(EcucModuleConfigurationValues\) is a type of OS Module-Configuration container as explained in chapter 3.1.3. This OS module configuration values stores OS relevant parameter values in form of container object with type of \(EcucContainerValue\). These objects are the required configuration aspects like OS application, Counter, Tasks etc as displayed in 5.2(b). OS application is an existing container (\(osApp\)) which accumulates all the related OS elements mentioned above along with the configuration process, but non-existing OS parameter objects need to be created. So to generate these OS elements, the algorithm needs valid information gathered in an object. Therefore, for every element that requires to be generated freshly, there are java classes written to encapsulate relevant information for these elements. Then, using object instance of the class, defined function will generate the actual module supporting container with configured value. OS counter which has the most important parameter information like \(SecondsPerTick\), which sets the resolution of OS schedule table iteration tick. This value is set to a fixed measure of 0.001. Naturally, the objective is to configure OS module, therefore other parameters (OS tasks, events) are missing that need to be generated and embedded to the module also.

Using Xtend Java command shown below, every change or creation during algorithm 6 will be applied to OS module.

\[
OS = \Rightarrow [\text{Algorithm 6}]
\]
The execution of tasks and eventually the execution of corresponding runnable is
invoked by using OS Events (E). Therefore, one OS event is necessary to be created
for each runnable using its short name for unique identity. All events are collected
in an event list e. Also, each event (E) is mapped to its corresponding runnable (R)
for better extraction in later part of algorithm as $R_E$.

As discussed in section 4.1, it is mentioned that to achieve graceful degradation,
a mechanism of system modes (L) is introduced. In each system mode, different
environment of failures are managed and configured by changing the location
of SWCs. A developed system should be scheduled for these configured system
modes, since each mode has remapping of SWC prototype. Therefore, completely
scheduled system will have all the scheduling requirements configured and encapsu-
lated in SystemTiming AUTOSAR object in the form of Concrete Pattern events.
These mode specific system timing elements can be collected by a function named
getScheduling from the scheduled system model (SSM). This function returns a list
of Scheduling Requirements (SR) object where each SR object holds system tim-
ing and its concrete pattern events (CPE), TDEventSwcInternalBehavior (TDE)
and calculated constant schedule table period. This list of SR is further used in
runnable to task mapping considering ASIL level algorithm, which is an important
part of the OS configuration. The algorithm 5 in chapter 5.2 gives an output as a
list named Tasks where each task (T)(string name with unique identity) is mapped
with a runnable group ($R_T$) belonging to T. Next, the OS configurator creates OS
5 Implementation

Algorithm 6 OS AutoConfiguration

1: Data: P, Δ, SSM
2: SR, STperiod, FlatMap ← empty
3: SR ← getScheduling(SSM)
4: STperiod ← getEcuFlat ECU_X
5: e, osTasks, osST ← empty
6: for each R in Δ do
7:   E ← new osEvent (R.shortName)
8:   e ← map(R, E)
end for
9: Algorithm1
10: Algorithm3
11: osCounter ← new OsCounter
12: osCounter.SecondsPerTick ← 0.001
13: osApp.counterRef ← osCounter
14: Algorithm5
15: for each T in Tasks do
16:   osTask ← new OsTask(T)
17:   for each R in RT do
18:     RE ← e(R)
19:   end for
20:   osTask.osApplicationRef ← textosApp
21:   osTasks ← osTask
22: end for
23: for each L in SR do
24:   CPE_L ← L.CPE
25:   ST ← new ScheduleTable (L.shortName)
26:   ST.period ← STperiod
27:   for each pattern in CPE_L do
28:     TDE_p ← pattern.TDE
29:     if TDE_p.SwcBehaviourType = Runnable_Entity_Activated then
30:       PSYS ← TDE_p.contextComponent
31:       RP ← TDE_p.runnable
32:       P_ECU ← FlatMap(PSYS)
33:       if P_ECU exist in P then
34:         OA ← pattern.Offset
35:         EP ← createExpiryPoint(OA)
36:         RE ← e(RP)
37:         ES ← createEventSetting(RE)
38:         EP.subContainers ← ES
39:         ST.subContainers ← EP
        end for
30:   osApp.STref ← ST
31: end for
32: os.containers ← osST, osTasks, e, osApp
5 Implementation

tasks for each task $T$ and concurrently for each OS task, runnable OS event $R_E$ (in such a way that $R_E \in R_T$) reference is given. These runnable OS events can be extracted from a list of OS events $e$. In this manner, every created OS task will have a runnable event ($R_E$) reference. Similar to the OS Counter, the $OsTask$ object encapsulates OS task regarding parameter details such as task name (same as $T$), event references, task stack size, OS application, priority and activation. Container holding all such information is created for each task and then configured accordingly. OS task and application should point to each other for declaration of their contract. Every created OS task container is accumulated in a list $osTasks$.

As discussed earlier in 3.1.4, scheduled system model (SSM) has system timing element holding timing description of each runnable execution time as CPE and TDE. These system timings are mode (L) specific encapsulated in scheduling requirement (SR) java object. According to the requirement, each mode should have a separate OS Schedule Table (ST), which is configured using L mode short name and the global constant $STperiod$. CPE has a pattern of type $ConcretePatternEventTriggering$ which has $MultidimensionalTime$ elements with timing values. Patterns of CPE reference to TDE which gives the runnable and system SWC prototype it belongs to (refer to 5.1). The purpose of this step is to find out initial activation offset value for each runnable belonging to the ECU. Therefore, an algorithm is only interested in pattern with activation value. Hence, there is a condition to filter $TDE_P$ with internal behavior of type $RUNNABLE_ENTITY_ACTIVATED$. Context component of $TDE_P$ is reference to system SWC prototype ($P_{SYS}$), but the program requires ECU perspective flat view prototype. This link between system abstract prototype $P_{SYS}$ and ECU flat SWC prototype $P_{ECU}$ is in ECU extract ($ECUX$) file in FlatMap AUTOSAR model object. Once found using link as shown in figure 5.1, it is checked across the list of all prototypes of ECU for its existence. If $P_{ECU}$ belongs to the ECU (current configuring one), then the program goes further for the creation of schedule table expiry points and event setting.

The required offset values from filtered patterns are obtained by $realTimeValue$ function using semantics of $cseCode$ and $cseCodeFactor$ (explained in 3.1.4) as double type value. Now, the obtained offset value is nothing but the activation offset ($O_A$) for the corresponding runnable ($R_P$) which is taken as the expiry point offset for the execution on schedule table. Therefore, Expiry Point (EP) is created for each offset value $O_A$ and also runnable OS event ($R_E$) is taken from list of OS events $e$. Event setting of expiry point EP is given the reference of runnable OS event ($R_E$) which is found before. All the ECU configuration containers created are gathered together hierarchically as shown in line 42. Os application is always linked to the new ECU configuration parameter value. Finally, accumulated OS events, Tasks, Schedule Tables, application is added to the sub containers of OS module, which will get updated in ECU Configuration file.
5 Implementation

5.3.2 RTE

From 3.1.1, it is clear that only relevant objects of RTE configuration are SWC prototype Instances and RTE event to task mapping. Hence, an algorithm is developed which handles the creation of such objects. Though just two objects, there are several referencing objects that need to be created and embedded to the RTE module.

When entities of Software-Components are to be configured, there is the need to actually address the instances of the AtomicSwComponentType. Since the ECU Extract of system description contains a flat view on the ECUs Software-Components. The SwComponentPrototypes already represent the instances of the Software Components. Furthermore, one of the major fragments of the RTE configuration is the mapping of AUTOSAR SWCs Runnable Entity to OS Tasks. Activation of Runnable Entity belonging to the mapped SWC on ECU is based on RTE Event, which is already created during SWC development. Therefore, there should be a mapping container referencing RTE Event for activation of Runnable Entity.

Similar to OS module, RTE module is a EcucModuleConfigurationValues type module object which basically holds the entire configuration parameter value, where type of these parameters are EcucContainerValue with reference to different definitions. Definitions of such containers reveal the purpose of the object. From Figure 5.2(a), selected sub container of RTE module named AccelerationIndicatorProto is referencing to the SWC prototype of the same name and it has definition object named RteSwComponentInstance, which tells it is an instance for the prototype. Now this Instance container contains its own parameter object as a value for the configuration e.g, sub container of Instance with definition of RteEventToTaskMapping, which itself explains that it handles the mapping of RTE Event to OS tasks. Similarly, there are multiple parameter objects for several purposes, but in the context of this thesis the only relevant parameter that has to be configured is RTE Event to the OS tasks. Hence, an Instance object for each SWC prototype present in the ECU extract and in that, configuration of Rte Event to OS tasks.

\[
\text{RTE} \Rightarrow \text{[Algorithm 7]}
\]

RTE configuration asks for creating RTE instances \((I)\) of all the SWC prototypes \((P)\) present in ECU with the instance definition referencing to specification object RteSwComponentInstance which is taken from Module-Definition of the RTE module. According to the chapter, this instance definition also includes Container-Def to specify the parameter type and specification for the sub containers of RTE instances “I”. Now, each RTE instance has a reference value of corresponding SWC prototype.
Algorithm 7 RTE AutoConfiguration

1: **Data:** osTasks, P, e

2: **for each** p in P **do**

3: \( I \leftarrow \) new RteSwComponentInstance\((p\text{.shortName})\)

4: \( P_{ref} \leftarrow \) new RteSoftwareComponentInstanceRef\((p)\)

5: \( I\text{.referenceValues} \leftarrow P_{ref} \)

6: \( R_P \leftarrow \) getRunnables\((p)\)

7: \( R_E \leftarrow \) e\((R_P)\)

8: \( RTE\text{Events} \leftarrow \) getRteEvents\((p)\)

9: **for each** event in RTE\text{Events} **do**

10: \( \text{if} \) event.type = OperationInvokedEvent **then**

11: \( Task \leftarrow \) empty

12: **for each** T in osTasks **do**

13: \( \text{refValues} \leftarrow \) empty

14: \( \text{refValues} \leftarrow T\text{.referencevalues} \)

15: \( \text{if} \) \( R_E \) exist in refValues **then**

16: \( Task \leftarrow T \)

17: **end for**

18: taskmap \( \leftarrow \) empty

19: taskmap \( \leftarrow \) createTaskMapSubcontainer\((\text{event}, \text{Task}, R_E)\)

20: I\text{.subContainers} \( \leftarrow \) taskmap

**end for**
of the ECU. Hence, a new ECU configuration container is created with definition reference to \texttt{RteSoftwareComponentInstanceRef} object from Instance definition and then this container is added to the reference values of the RTE instance. Further, runnable belonging to SWC prototype \( P \) is extracted using \texttt{getRunnables} function in \( R_P \). The runnable OS events \( e \) list is obtained from the previously configured OS module where OS events exist for each runnable present in ECU including \( R_P \). Hence, runnable OS event for \( R_P \) is obtained from events list \( e \). Also, there is a need for RTE event list of SWC prototype which is extracted by using a function called \texttt{getRteEvents}. Now, for each RTE event of type \texttt{OperationInvokedEvent}, available data is runnable \( R \) and runnable OS event \( R_E \). Algorithm finds task from OS task container \texttt{osTasks} which reference to \( R_E \) and store it in \texttt{Task}. At this time, available data is \( R_E \), RTE event, Task. Using these references and available data, algorithm creates Rte Event to OS tasks container with definition specification of \texttt{RteEventToTaskMapping} using function called \texttt{createTaskMapSubcontainer}. This container gives reference to available data of \( R_E \), RTE event, Task. Finally, created container is added to the sub-containers of RTE instance \( I \) created earlier.

5.4 Generation of Generic Reconfiguration Plans

As discussed in 2.2.2, a module providing monitoring and reconfiguration services is developed with pre-defined adaptation plan for transition from one system mode to another. This chapter gives an implementation of a generic adaptation plans creator required in this module.

At the beginning, the program has to configure initialization mode manually, so the first transition is always from initialization mode to Normal mode, but in this thesis context we are assuming that system is already in a Normal mode, hence the algorithm is written in assumption that initial state is Normal mode. In normal mode, all the resources are considered available, which means that every ECU application is functional. For the sake of simplicity, let us take the previous example in which system has only three number of ECUs (ECU1, ECU2 and ECU3). So, Figure 5.3 shows the same mode adaptation flow as a generic one from Figure 4.10 but specifically for limited number of ECUs. As shown in Figure 5.3, in normal mode (M0) all three ECU are available, which will be the initial mode for this algorithm. Then, next transition of mode is possible when one or more ECUs are unavailable. Therefore, possible combinations of failure situation (M1,M2...M6) modes are determined and adaptation plans has to be created for transition to each of these failure modes. Considering the failure mode as child mode and normal mode as parent mode since the transition is from normal mode as shown in the Figure 5.3. Now, once this hierarchy of transition plans are created, the algorithm goes to the next level of adaptation hierarchy in which the child modes (M1,M2...M6) of the previous level will act as parent mode and the program creates adaptation plans for their possible failure transition. This cycle of plan creation continues until there are no
5 Implementation

possible ways of system failure.

Figure 5.3: Adaptation Algorithm Flow

Following steps are taken for this heuristic:

I Collect resources, application prototype, configured modes, scheduling requirements.

II Find software component prototypes of each ECU linking them together; create Application Instances for each prototype with default state (DEACTIVATED) and Control Unit Configuration for each ECU present in the resources.

III Gather all possible failure combination of resources from active mode, where each combination will be stated as child mode of active mode (parent mode). Then, create an adaptation plan for each transition from parent mode to child mode by using breadth search algorithm explained in 4.3.

IV Once adaptation plans are created for every possible parent to child mode transition, treat every child mode found in step III as parent mode and perform step III again until there are no possibilities of child mode.

So, for algorithm to go through these steps, it needs to have information about parent mode and child mode. For this purpose, there is an object called “Adaptation Owner”, which keeps the information about these two parts and updates it whenever
the algorithm switches from one hierarchy level to another. Initially, required data which are relevant for this algorithm must be collected from the well-planned system model. Extracting such an information is possible due to the support and interface of the ARTOP framework, which basically allows the Eclipse plug-in project to access all the elements present in the “arxml” system model file. All the resources (ECU instances \( E = \{ E_1, E_2...E_n \} \) collection and the software component prototypes \( P = \{ P_1, P_2...P_i \} \) belong to the ECU are mapped together such that \( P \) belongs to \( E \), for better extraction in the later stage. Also from scheduled system model, the extraction of all system modes \( \delta = \{ M_1, M_2...M_y \} \) which is stored as ModeDeclaration AUTOSAR model object and system schedules \( S_{M_y} \) for each mode is done. According to Step III, this algorithm needs an adaptation plan creator for current parent mode which takes a child mode and creates plan for the corresponding transition.

Adaptation plans creator is an algorithm which takes a group of resources as an argument and, by knowing the information about the adaptation owners, it creates transition plans for the corresponding adaptation. Since system modes \( (\delta) \), parent mode \( (M_p) \) resources are global variables of this class, they are visible to this creator algorithm, which is a vital information. So, this algorithm treats passed group of resources as a child mode \( (M_c) \) with available resources and knowing the \( M_p \), it creates adaptation plan \( (AP) \) from parent to child mode. Later, these adaptation plans are collected by mapping them with their owner \( (M_p, M_c) \) in \( \rho \), where \( \rho = \{ AP_{O_p,1}, AP_{O_p,2}...AP_{O_p,t} \} \). In an adaptation plan object, there are two lists of prototype instances \( (I) \) with their instance state \( (S) \), where these two list are the “Current State” and “Target State” of the system for this adaptation plan. So, an adaptation plans basically makes transition from current state of system \( (AP.currentState) \) to target state of system \( (AP .targetState) \) which is the required state. The algorithm determines the unavailable or failed resource \( (E_{failed}) \) by comparing resources available in \( M_c \) (\( E_c \) such that \( E_c \in M_c \)) to parent mode resources \( (E_p|E_p \in M_p) \) such that \( E_{failed} = E_p - E_c \), and collect instances \( (I_{failed}) \) of prototypes belonging to failed resource. Therefore, in the current state of the system, \( I_{failed} \) are failed instances and should be given state as DEACTIVATED. To determine the state of instances in a particular mode, the algorithm needs configured system mode \( (\delta) \) data which encapsulates prototype and their state. This mode configuration \( (M_{config}) \) is extracted from collection of pre-configured modes by matching the resources \( (E_c) \) available in \( M_c \) with resources in modes from \( \delta \). Once the required system mode \( (M_{config}) \) is found, it accumulates prototypes \( (P) \) for each ECU \( (E) \) and gets the instances \( I \) belong to each \( P \) and also their instance state \( S \). These found instances and state should be the target state of applications, hence it will be stored in target state of AP. At the same time, current state of applications \( (AP.currentState) \) except failed instances \( (I_{failed}) \) will be the same state as their previous target state of adaptation plan as shown in figure 5.3. Therefore, the algorithm first finds out from a collection of already created adaptation plans \( (\rho) \), which one has parent mode \( (M_p) \) of currently processing adaptation plan \( (AP) \) as a child mode. All previously
Algorithm 8 Adaptation Plan Creator

1: Input: $M_c$
2: Data: $M_p, \delta, \rho$
3: $n \leftarrow \text{sizeof}(ASIL_X)$
4: $O \leftarrow \text{newAdaptation Owner}$
5: $AP \leftarrow \text{newAdaptation Plan}$
6: $O.parent \leftarrow M_p$
7: $O.child \leftarrow M_c$
8: $E_{\text{failed}} \leftarrow E_p - E_c$
9: $I_{\text{failed}} \leftarrow \emptyset$
10: for each $I \in E_{\text{failed}}$
11:   $AP.currentState \leftarrow \text{state}(I, \text{DEACTIVATE})$
12: $M_{\text{configured}} \leftarrow \delta(M_c)$
13: for each $E$ in $M_{\text{configured}}$
14:   $P \leftarrow \text{extractProto}(E)$
15:   for each $P_i$ in $P$
16:     $I \leftarrow \text{getInstance}(P_i)$
17:     $S \leftarrow \text{getState}(P_i)$
18:     $AP.targetState(I, S)$
19: end for
20: for each $AP_{O_{pc}}t$ in $\rho$
21:   $O_{\text{local}} \leftarrow \text{getOwner}(AP_{O_{pc}}t)$
22:   if $O_{\text{local}}.child = O.parent$
23:     for each $I$ in $AP_{O_{pc}}t.currentState$
24:       if $I \neq I_{\text{failed}}$
25:         $AP.currentState \leftarrow AP_{O_{pc}}t.targetState$
26:     end for
27: end for
28: $\rho \leftarrow \text{map}(AP, O)$
29: return $\rho$
5 Implementation

created transition plans are gathered in an object \( \rho \) by mapping adaptation plan and their owner together \( (AP_{O_{pc}}) \). Accordingly, algorithm checks for all adaptation plans in \( \rho \) across their owners \( (O_{local}) \). If the child mode of \( O_{local} \) matches with the current parent mode \( (M_p) \), then that is the required plan. Then, as per the requirement all the target states of \( AP_{O_{pc}} \) will be the current state of AM except the failed instances \( I_{failed} \). Therefore, another for loop checks every instance in \( AP_{O_{pc}} \) across \( I_{failed} \), so that only available instances state will get updated in the current state of AP. At the end, created plan \( (AP) \) is mapped with the owner \( (O) \) and collected in \( \rho \), which is a global variable.

Algorithm 9 carries out steps I, II, some part of III and IV to manage the adaptation process. Schedule table duration \( (\tau) \) is an important parameter, which is constant in this case for all the modes and determined from system scheduling \( S_{M_p} \). Therefore, even a single pattern period from one of the mode scheduling is enough, which can be extracted by the getScheduling function. This algorithm asks for Instances for each software component prototype present in each ECU, both are mapped together in list \( I_P \) which later become easier to extract. This is done by using a for loop between line 8 and 11 as shown in algorithm 9. ECU is a control unit configuration object, which encapsulates all the instances mapped to their states. So, as discussed earlier, theoretically this module needs initialization steps where system makes first adaptation from initial mode to normal mode but for simplicity this algorithm considers normal mode as the initial mode of system. As mentioned before, it is clear that to create new adaptation plan to new child, it also needs previous adaptation plan to inherent some target state. Hence, this algorithm creates initial adaptation plan in which the normal mode is parent and child mode at the same time. Therefore, initially when all the resources are available in the normal mode, they are treated as active parent mode \( (M_p) \) and active child mode \( (M_c) \), and the algorithm creates an adaptation plan for this using algorithm 8. So, now, algorithm has base adaptation plan in \( \rho \) which can be used further for next child mode transition.

Once the algorithm create a base adaptation plan for normal mode resources, it finds all the possible failure combinations of resources in current active mode. Each system modes has a number of resources available in it, which are a list of ECU instances \( (E) \). Basically, failure combinations of ECU instances within this list are determined such that maximum number of resources available \( (j) \) in each combination is less than the number of resources \( (n) \) in its parent mode. Therefore, the formula derived to determine the number of possible failure combinations \( (F_n) \) is as follows:
5 Implementation

\[ F_n = \sum_{j=1}^{n-1} ^nC_j \]  

where \( ^nC_j = \frac{n!}{j!(n-j)!} \)

Taking the formula, if we consider the previous example in which we had 3 ECUs in the system, then from the normal mode, the next possible number of transitions are 6, where \( j = 1, 2 \). The value of \( j \) tells you the number of available resources. Considering our example, the possible number of combinations with 2 ECUs available is 3 and similarly to have only 1 ECU available is 3, in a total of 6 combinations. These failure combinations are generated by the function called \( \text{combinations} \). The outer loop of the nested for loop from line number 20 to 26 in algorithm 8 executes the formula, in which it finds every possible failure combination of the resources. These combinations are then stored in a list named \( \sigma \), since each combination of resource group belongs to one mode (\( M_c \)). Now, for each of these modes, the algorithm creates adaptation plan as child mode by knowing \( M_p \) by using algorithm 8, which is inner loop between line 19 and 21 of algorithm 9. While creating adaptation plans for \( M_c \), it is also pushed to the linked list (AM) created on line 12, which is later used by the algorithm to generate lower hierarchy level adaptation plans. This is done by updating \( \Lambda \) at the end of the while loop between 15 to 22. At the same end, AM linked list pops each mode encapsulating their resource group, which in next iteration is treated as the parent mode. Since the AM is a linked list, it follows First In First Out (FIFO) queuing mechanism which is the requirement in the algorithm. In this fashion, all the possible child modes are eventually treated as a parent mode so that the next possible failure mode transition plans can be created. This is done until there is no possible failure mode transition and AM is empty, which is nothing but a while loop condition.
Algorithm 9 Adaptation Manager

1: **Data:** $E, P, S_{M_0}, \rho$
2: $I_P, M_P, \tau \leftarrow \text{empty}$
3: $\tau \leftarrow \text{scheduling}(S_{M_0})$
4: **for each** $E$ in $\delta$ **do**
5: \hspace{1em} $ECU \leftarrow \text{createControlUnit}(E)$
6: \hspace{1em} $P \leftarrow \text{getProto}(E)$
7: \hspace{1em} **for each** $i$ in $P$ **do**
8: \hspace{2em} $I \leftarrow \text{createInstances}(P_i, ECU)$
9: \hspace{2em} $I_P \leftarrow \text{map}(I, P_i)$
10: **end for**
11: $\Lambda \leftarrow E \in \delta(M_c)$
12: $n \leftarrow \text{sizeof}(\Lambda)$
13: $A_M \leftarrow \text{LinkedList}(\text{empty})$
14: **while** sizeof($\Lambda$) $\neq 1 \&\& A_M \neq \text{empty} \text{ do**}
15: \hspace{1em} $M_P \leftarrow \Lambda$
16: \hspace{1em} **if** $AP \text{ exist in } \delta \ || \ AP.M_c = M_p \text{ then**}$
17: \hspace{2em} Algorithm 8($M_c$)
18: \hspace{1em} **for** $i = 1..n - 1$ **do**
19: \hspace{2em} $\sigma \leftarrow \text{combinations}(\Lambda, i)$
20: \hspace{2em} **for each** $M_c$ in $\sigma$ **do**
21: \hspace{3em} $A_M.push(M_c)$
22: \hspace{3em} Algorithm 8($M_c$)
23: **end for**
24: \hspace{1em} $\Lambda \leftarrow A_M.pop$
25: **end while**
6 Evaluation

The goal of this section is to give a proof of concept by means of the case study. In order to achieve this goal, an implementation presented is evaluated on a test bench based on a Demonstrator car architecture.

6.1 Setup

As mentioned in 3.2, for the purpose of software tooling in this thesis a “plug-in” projects of Eclipse platform is utilized to create new feature of an automatic configurator. This section gives a brief introduction on a system model and test bench platform used for the case study.

![Demo-Car System Model](image)

Figure 6.1: Demo-Car System Model

6.2 Case Study

In this chapter, the proposed algorithms is tested by using specific use case system model of a Demonstrator car. A system model configured with all modes com-
positions and schedules information is passed through the automatic configurator presented in this paper. At the end, configured system model is evaluated on the available setup of software tools and hardware platform. The next section explains the setup used for the proof of concept proposed during the thesis work.

6.2.1 System Architecture

A Demonstrator car system model is developed during safe adapt [19] as shown in figure 6.1 with the software instances of SAPC which provides a monitoring and reconfiguration service. This system model consisting of two critical e.g. Steering, Braking and one non-critical application e.g. Acceleration Indicator SWC prototype was developed in dSapce system desk modeling tool. Each critical SWC has two instances, one for normal mode and other is degraded instance for failure modes. To synchronize SAPC module SWC instances with the ECU execution cycle there is another synchronization SWC prototype for the each ECU. Also, some sensor and actuator type SWC for Cockpit exists to sense the action such as braking, steering or acceleration. Finally, a display SWC prototype for simulation of this control system. This system model is developed in consideration of hardware test bench for Demonstrator car shown and explained in 6.2.2.

According to design process from 4.1, a developed system model goes through each process stage and at the configuration stage scheduling parameters are generated by the automatic configurator (work of this thesis) for scheduling related AUTOSAR ECU modules.

A software development using plug-in project allows developer to test programmed feature on the another instance of Eclipse as mentioned in 3.2. Such a kind of cross platform testing helps the developer to debug the programmed feature.
6 Evaluation

From block diagram in 4.5, the automatic configurator has three input files which is utilized to configure the parameters of OS and RTE modules. After configuring parameters, modules are written back to the same input ECU configuration file. For the purpose to differentiate generated file from original one, output is written on the different file with same name but additional extension of “.arxml”. As shown in figure 6.2(a) and 6.3(a), the input file without configured parameters of OS and RTE modules.

Demonstrator car 6.2 and 6.3 are screen shots of Arctic Studio ECU configuration tool. Generated modules from automatic configurator is evaluated across this tool by using validation feature of Arctic Studio. As it is easy to differentiate input and output file from 6.2 and 6.3, where new containers and reference values has been added. From figure 6.2(b), additional containers for different OS tasks, events and Schedule tables are generated in module with some configuration values. Similarly as shown in 6.3(b), new instance containers has been added and mapped to the generated OS tasks and events.

6.2.2 Test Bench

After generating source files for corresponding control units, they are flashed into ARTiS (Automotive Real-Time Prototyping System) ECU platform (“ARTiS-RT, ARTiS-XT”) in Demonstrator car hardware setup shown in figure 6.4. In this setup, a prototype car with two ARTiS platform is used with external power supply, steering, pedals for braking and acceleration. Also, two desktops with a Windows and Linux operating system are present to monitor messages on Controller Area Network (CAN) communication bus. A failure control functionality integrated into this setup enables the test engineer to make one of the ECUs unavailable by means of cutting the power supply to ECUs and therefore fail-operational behavior can be analyzed with bus monitoring desktop and simulation on “Monitor” as shown in figure 6.4.
6 Evaluation

6.3 Fail-Over Time Analysis

As stated in system architecture, Demonstrator car model has software component instances for SAPC. During the run-time execution, monitoring and reconfiguration service is provided by these components. Figure 6.5 is given to visualize the real-time execution of system embedded with SAPC and to understand the worst-case scenario in which fail-over time will be maximum.

To calculate the maximum fail-over time of the system, two worst-case scenarios have to be considered. The first case, in which we assume that failure of ECU occurs just after execution of SAPC and before the execution of the critical software component. This failure is only detectable at the next execution point of SAPC. This situation makes period of the SAPC component fail-over time of the system if the critical component is immediately available for execution in failure mode schedule.

As per 2.2.2, every failure or system modes have their own schedule and hence their own scheduling table for execution. The second worst case will be the maximum offset a critical component has in the schedule table of switched failure mode. Therefore, considering this and worst case from above, the maximum worst-case
### 6 Evaluation

#### System Modes

<table>
<thead>
<tr>
<th>Available ECUS</th>
<th>Real-Time Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Normal Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU1 SAPC NCF</td>
<td>SAPC NCF</td>
</tr>
<tr>
<td>ECU2 SAPC</td>
<td>SAPC</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU2 SAPC</td>
<td>SAPC</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

#### Global Time

- **ST_normal**
- **ST_failure**

---

**Figure 6.5: Fail-Over Analysis**

fail-over ($F_{t\text{worst-case}}$) time for the system should be the sum of the period of SAPC ($P_{SAPC}$) software component and maximum offset ($O_{\text{max}}$) of critical component in the schedule table of failure mode. This whole scenario is portrayed in the figure 6.5.

$$F_{t\text{worst-case}} = P_{SAPC} + O_{\text{max}}$$

**Observation** A simple Demonstrator car system's control unit source files are flashed into the test bench platform as mentioned in 6.2.2. In order to test the fail-operational behavior of system, a failure control functionality is integrated in the test bench. This enables the developer to test mode transition parameters are configured correctly or not. Discussion from 6.3, the worst-case fail-over time for the used system model is 30 ms theoretically, as period SAPC component is 10 ms and steering prototype maximum offset is 20 ms in all failure mode. But practically observed value for the fail-over time is 13 ms (Note: Not in the worst-case scenario, as it is difficult to induce failure exactly at the point of worst case scenario).

#### 6.4 Discussion

It is observed from 6.3, that worst-case fail-overtime is dependent on SAPC frequency and the offset of a critical component in schedule table of failure mode. Therefore, one of the ways to improve the fail-over time will be to reduce the period of SAPC, but by doing so overhead of the system increases. For this reason, there will always be a trade-off between system overhead and fail-over time. The another way will be to take care of critical component offset during scheduling which
6 Evaluation

is not possible as in failure-mode all the component are critical one. Hence, further research on this problem is necessary in order to improve the fail-over time.
7 Conclusion and Outlook

In order to achieve fail-operational behavior in automotive domain, which is on the verge of evolution of autonomous driving, a design process is developed on the basis of AUTOSAR. This design process needs a support of automated service during control unit configuration stage. To fulfill this requirement, this paper have presented an approach to automate the process of configuration and, also a method to generate system mode transition plans automatically for AUTOSAR based system.

Since the work of this thesis is tooling based, several paper on AUTOSAR tooling [32, 30] have helped to understand the data structure and exchange methodology. The implementation considers the criticality level of functions and the run-time execution preemption. In the past, multiple papers [16, 43] have been submitted to handle such a mixed criticality problem. However, these approaches were found unsuitable for the task of this thesis, where mixed criticality is considered for scheduled system. Nonetheless, it was helpful for understanding the topic of mixed criticality. To handle the AUTOSAR specific runnable conflict problem, this thesis have used multiple theoretical concept such as register conflict graph and graph coloring method from the academic studies. It is found that even if the greedy algorithm for graph coloring is the easiest solution for the runnables partitioning, it is not the best solution. Alternative for the greedy approach is Welsh theory as mentioned in chapter 4.2. Furthermore, to navigate through the possible system modes to generate reconfiguration plans, an appropriate search algorithm was necessary. For that purpose, several search algorithms were referred for the implementation, from which Breadth search algorithm (chapter 4.3) was found to be relevant to this topic.

Using conventional method for the configuration of scheduling parameters for system model formed with several operation modes was found difficult, inefficient and full of human error. An addition of such an automated service overcomes the problems faced by conventional method. Also, a method is proposed to generate deterministic adaptation plans for the system which improves system's predictability on the occasion of control unit failure. Concurrently, this method synthesized an external module providing monitoring and reconfiguration service as a part of AUTOSAR service layer.

A solution proposed in this work to handle Mixed criticality is still not the optimum solution considering the constraints (same memory for one task's jobs) of a processor in computing unit. Also, it is possible to replace the conventional method
of system configuration by this automatic approach by exploring possible advantages in further studies. Finally, further research is required to improve fail-over time of the system having reconfiguration module.
Bibliography


[16] Christoph Ficek, Nico Feiertag, Dr. Kai Richter: Applying the AUTOSAR Timing Protection to Build Safe and Efficient ISO 26262 Mixed-Criticality Systems


[23] Fleming, T.: Extending Mixed Criticality Scheduling. mscreserch, University of York (Sep 2013)


BIBLIOGRAPHY


[52] Yang, Y.: Software Synthesis for Distributed Embedded Systems. eScholarship (Jan 2012)
