Modeling the Effects of AUTOSAR Overhead on Automotive Application Software Timing and Schedulability

by

Manish Chauhan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

Abstract

AUTOSAR (AUTomotive Open System ARchitecture) provides an open and standardized E/E architecture to support modularity, transferability, reusability and scalability of the various components required to implement a function in a vehicle. AUTOSAR has become the de-facto standard for the automotive application development. Safety-critical nature of the automobiles makes the automotive application development challenging, and due to the growing complexity of the software in modern day vehicles, it has become even more challenging. A system is called schedulable when it meets all its real-time requirements under all the possible scenarios. An automotive application should always be schedulable; failing it can have grim consequences. The overhead added by the AUTOSAR stack can significantly change the schedulability of an automotive application. This thesis proposes an overhead-aware method to find a schedulable design configuration for an AUTOSAR application. The method allows measuring the overheads of an AUTOSAR stack implementation and assessing the impacts of the overheads on the timing and schedulability of an application using a timing model of the application. The thesis demonstrates the application of the method on a case study, and finally, it demonstrates the effects of the different types of system overheads on the timing and schedulability on a range of synthetic applications.

Acknowledgements

I would like to thank all the people who have supported me throughout the journey. I am grateful to my supervisors Prof. Krzysztof Czarnecki and Prof. Rodolfo Pellizzoni for their guidance and encouragement.

I would also like to acknowledge the ARCCORE support staff for their help during the AUTOSAR application development.

Dedication

This thesis is dedicated to my grandfather and my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The software applications used in the automotive industry are very complex. These applications have high safety requirements, stringent timing constraints and are required to interact with many other software applications and hardware, e.g., ABS (Anti-lock Braking System), cruise control application. In a vehicle, the effects of a task missing its deadline can be catastrophic. For example, in ABS application, if the task responsible for applying breaks misses its deadline the breaks of the vehicle will not be applied. This unintended behavior of ABS applicaion can lead to severe consequences and create a safety threat. It is of utmost importance that all the applications in a vehicle are always schedulable and perform the functionalities as expected. A system is called schedulable if the execution of all the jobs of all its tasks is guaranteed to be completed before their respective deadlines. Schedulability analysis is the technique used to know whether the application is schedulable or not. If the schedulability analysis of the system cannot guarantee that system is schedulable then system is considered unschedulable. The schedulability of the system depends on many factors, e.g., task running time, task deadline, operating system, and hardware.

AUTOSAR is a worldwide partnership between major OEMs (Original Equipment Manufacturers) and tier-1 suppliers to standardize an open software E/E architecture for automobiles [5]. This standardization eases the process of automobile application development and provides other benefits such as transferability, scalability, maintainability, safety, lower development time and cost. The design process of an AUTOSAR application usually starts from informal specifications about the functionality and their time bounds; more details are gradually added to it. To keep time and cost low, it is important to check at every stage of application development whether the application is schedulable or not. Finding unschedulability of the application in early stage of its development can save much of time and efforts. However, estimating the performance of the application in the early stage of its development is limited by the information available about the whole system. This lack of information decreases the accuracy of early stage estimates. The application design phase is one of the early phases of application development. Generally, design phase is the earliest phase when one has sufficient information to check the performance parameters of the application. Furthermore, the performance estimation of the application in this phase will be advantageous for the designer as they can explore different design alternatives and even choose the hardware required [26].

AUTOSAR applications run with the infrastructural support provided by the AUTOSAR stack. This stack is generated based on the application needs and provides all the interfaces to access hardware devices and other system services, e.g., communication services, and OS services. All these facilities come with overhead cost added by the AUTOSAR stack. Unfortunately, this overhead is generally overlooked during the design process and performance testing in the early stages of the application development. This overhead can be significant enough to have effects on the AUTOSAR application's performance. Not considering these overheads in the performance estimation can result is poor performance of the application to the complete failure of it. This is the problem targeted in this thesis.

1.1 Contributions

The main contributions of this thesis are as follows:

1. An overhead-aware method for finding the schedulable AUTOSAR software application design configuration. The proposed method can also be used for performance estimation and design exploration of an AUTOSAR application.
2. Modeling the impacts of different types of system added overheads using synthetic AUTOSAR applications.

Other contributions of this thesis are as follows:

1. Quantification of the different types of overheads due to the AUTOSAR stack
2. A framework to generate synthetic AUTOSAR applications
3. A framework to measure the execution time of any part of the AUTOSAR application or stack implementation

1.2 Related Work

Schedulability and performance analysis have been researched extensively. Over the years, many techniques for performing schedulability analysis have been developed, e.g. offset-based and holistic [31][32]. Since testing methods cannot guarantee that the worst case is tested, they are insufficient to check if timing requirements of the application are met in worst case conditions. This is the reason mathematical methods for schedulability analysis are used. These mathematical methods are implemented using software tools to do the schedulability analysis of an application. The most popular open source schedulability analysis tools are MAST and Cheddar [20][14]; they both offer standard feasibility testing, including the schedulability analysis of fixed priority and EDF based systems. One important difference between these two is that MAST can perform function level characterization, while Cheddar is limited to only task level characterization of the application. For this reason, our proposed method uses MAST.

In the context of timing and schedulability analysis of AUTOSAR, the work done is not extensive. For time modeling and analysis of AUTOSAR applications, TADL2 (Timing Augmented Description Language) has been developed under the TIMMO-2-USE project [30]. TADL is capable of symbolic timing expression modeling, timing constraint specifications, and probabilistic timing information. It is aligned with AUTOSAR timing models. TADL is used to add the timing information to an AUTOSAR model which completes the model for timing and schedulability analysis. Later versions of AUTOSAR introduced timing extensions to specify the timing requirements of the application [6]. Hladik, et al., have analyzed a subset of AUTOSAR OS specifications from the schedulability perspective [33]. They concluded that the timing protection and schedule tables AUTOSAR OS extensions correspond to the pre-existing concepts of real-time scheduling theory. Also, existing methodologies of the real-time scheduling theory can be used for analysis of real-time AUTOSAR applications, mainly because these applications essentially use the scheduling services provided by the AUTOSAR OS, which is implemented by following the specification studied. Anssi, et al., have presented an approach for scheduling analysis of the AUTOSAR applications [29]. They have identified the main components of a model required to perform the scheduling analysis on it. They have also shown how these requirements are met by an AUTOSAR model. For the schedulability analysis, they have used MAST. In their analysis, the workload consisted only of the application execution time and overhead added by the AUTOSAR stack was not considered.

The closest attempt to observe the effects of overheads on the schedulability of an AUTOSAR application is by [24] and [22]. Mehdi, et al., have discussed the overhead of an RTOS. Their study is focused on how the RTOS overheads change if more than one

RTOS is hosted on an ECU [24]. Moghaddam has performed a study for communication overheads, focusing mainly on comparison of communication overhead in the inter-core and the intra-core communication [22].

1.3 Thesis Organization

The rest of this thesis is organized in four chapters. Chapter **2** describes the background information necessary for understanding the thesis. This chapter provides all the required details about AUTOSAR and MAST. **Chapter 3** presents details about the proposed method and all its steps. **Chapter 4** first evaluates the proposed method using a concrete use-case to show the feasibility of the approach, and then later generalizes the effects of the AUTOSAR system overheads on the schedulability of an AUTOSAR application using synthetic applications. **Chapter 5** presents the conclusion of the thesis followed by the future work.

Chapter 2

Background

This chapter gives the overview of the technologies and other concepts required to understand the following chapters. The following sections present the details about the AUTOSAR standard and the MAST tool, used for the schedulability analysis in the proposed method.

2.1 AUTOSAR

AUTOSAR (Automotive Open System Architecture) is a consortium founded in 2003 by the OEM manufacturers and leading automotive suppliers. The target of this alliance is to collaboratively standardize the open industry standard for automotive E/E architecture to overcome the growing complexity of software in present-day vehicles while still leaving room for competition by innovative implementations. In other words, the motto of the AUTOSAR consortium is "Cooperate on standards, compete on implementation" [5].

The AUTOSAR standard is specified by a set of specifications. These standard specifications are documents, models or templates which specify the normative results of AUTOSAR partnership [5]. The areas these specifications cover are:

1. Software Architecture including software applications, the environment in which applications will run, and the entire stack running on an ECU which is commonly called BSW (Basic Software).
2. Methodologies for creating SWCs (Software Component) and basic software modules [10][7]. For all of these, it also describes the description and exchange formats for integration, and guidelines for using the framework.

3. Syntax and semantics for all the interfaces of automotive applications from all the domains.

2.1.1 Benefits of AUTOSAR

A product has to fulfill all the related requirements specified by the AUTOSAR standard to become AUTOSAR compliant. The benefits of becoming AUTOSAR compliant are:

1. Maintenance, updates, and upgrades become easy because of the standardized specification. As shown in **Figure 2.1**, before AUTOSAR there was no clear separation between application software, platform, and the lower layer hardware dependent code. AUTOSAR standard has provided the interface which abstracts all the layers of the Stack from each other.

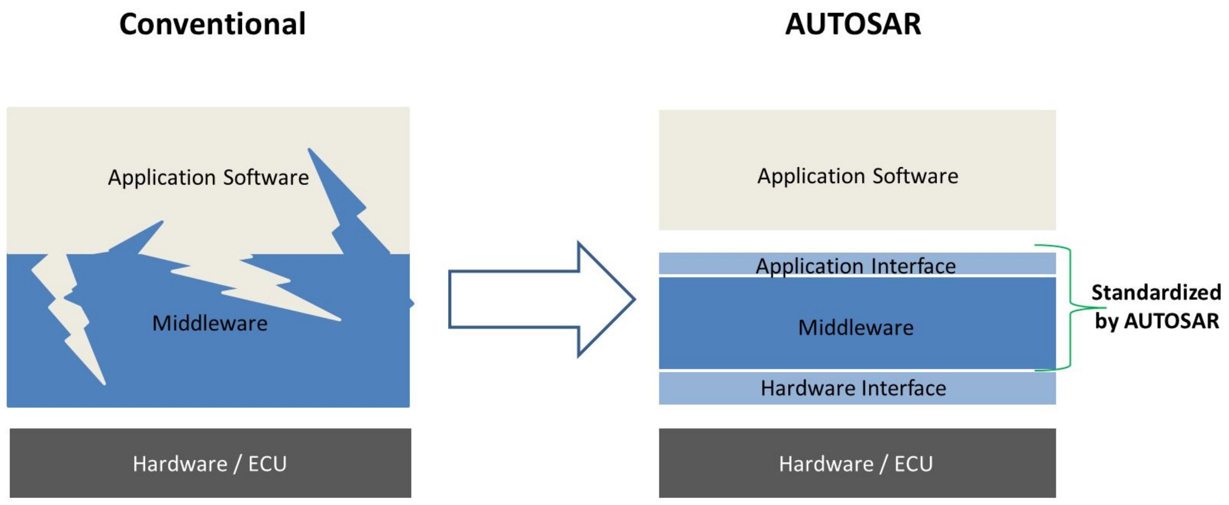


Figure 2.1: AUTOSAR provides abstraction between different layers of stack.

2. A module implementing a functionality can be used with different vehicle and variants of the platform. This reduces the version proliferation and increases the scalability of the solutions.

3. Integration of different product used in a vehicle becomes easier and better manageable as all interacting products share the same standardized interfaces.

4. By following the simple process and methodologies specified by the standard, products can be developed within lesser time and cost.

5. Having everything standardized creates room for more opportunities for new and small suppliers. This, in turn, gives more choices to the vehicle manufacturers.

2.1.2 Architecture

AUTOSAR has a layered architecture. The whole stack is conceptually divided into three layers - Basic Software Layer, RTE Layer, and Application Layer. These layers provide a high level of abstraction. Following sections discuss each of these layers in detail.

2.1.2.1 Basic Software Layer

BSW (Basic Software) layer provides all the infrastructural service to higher layers [7]. In itself, it does not implement any application functionality but provides the support needed by the upper layers to execute these functionalities. This layer contains standard and ECU specific modules. As shown in **Figure 2.2** Basic Software Layer is further divided into three sub-layers Microcontroller Abstraction Layer, ECU Abstraction Layer, and Service Layer.

2.1.2.1.1 Microcontroller Abstraction Layer

MCAL (Microcontroller Abstraction Layer) is the lowest layer in the AUTOSAR stack that provides the microcontroller level abstraction, thus making upper layers microcontroller independent. To provide this abstraction, MCAL manages microcontroller peripheral and provide underlying hardware independent value to the Basic Software layers above it. MCAL contains internal drivers to directly access the hardware which makes it hardware dependent, e.g., digital IO driver, serial peripherals.

2.1.2.1.2 ECU Abstraction Layer

ECUAL (ECU Abstraction Layer) decouples the higher BSW layers from the ECU by providing a software interface for accessing ECU specific resources. As a result, layers above ECUAL have no dependency on the ECU. This layer provides APIs to the higher layers to access all the available peripherals and devices regardless of their location. E.g, the application accessing a sensor will not know if the sensor is present on the same ECU or on a different ECU.

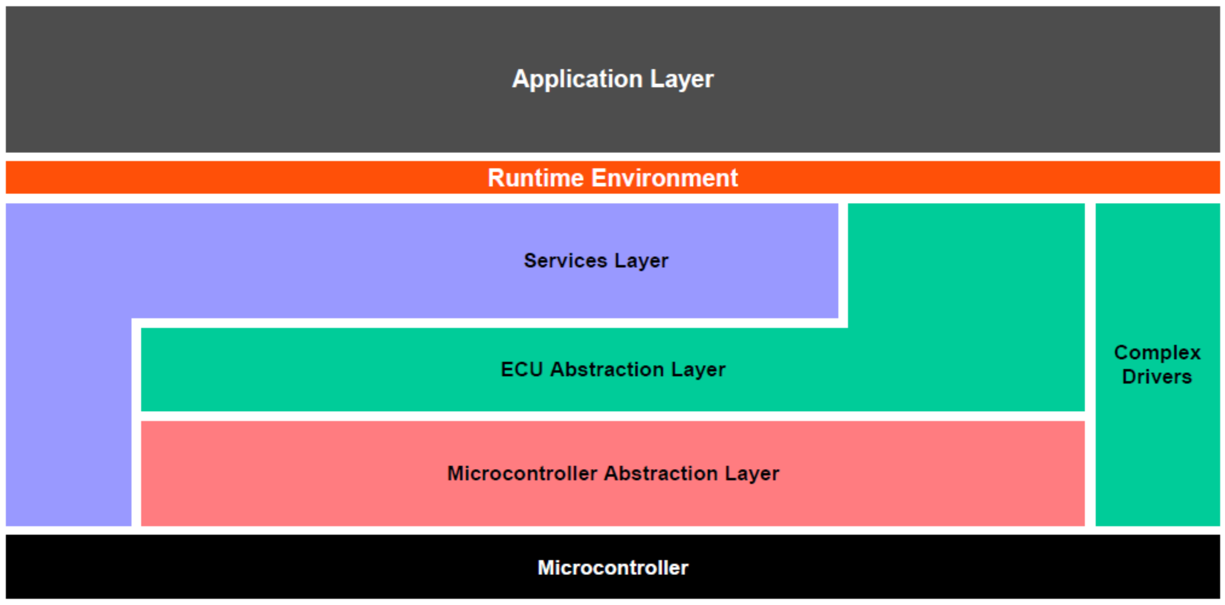


Figure 2.2: AUTOSAR Layered Architecture. Service Layer, ECU Abstraction Layer, and Micro-controller Abstraction Layer are part of Basic Software Layer [5]

2.1.2.1.3 Service Layer

SL (Service Layer) is the highest BSW layer which makes the RTE layer completely hardware-independent. This layer generally contains hardware independent code and provides the following functionalities:

- Operating System Services
- Memory Services
- ECU state and other related services
- Vehicle wide communication and resource related services
- Diagnostic and watchdog related services

2.1.2.1.4 Complex Device Drivers

CDD (Complex Device Drivers) spans through the entire BSW layer, i.e. from hardware to RTE layer. They provide room to add any additional functionality such as:

- Functionality with very tight time bounds which require direct access to hardware, e.g., complex drivers accessing microcontroller specific interrupts to access intricate sensors and actuators.
- Functionality for which AUTOSAR standards are not available.
- Functionality which is integrated with legacy architecture into the current system. This also helps in migrating legacy systems to AUTOSAR standards.

CDD generally contains hardware dependent code and also does not necessarily provide abstraction from hardware. So, an application using CDD could be hardware dependent and thus not AUTOSAR compliant, e.g., injection control, electric valve control.

Broadly, BSW Layer's services can be divided into four categories – Input/Output, Memory, Communication, and Systems. **Figure 2.3** shows the different modules implementing the BSW services at its three layers. IO services abstract the location of the peripheral IO devices and hardware layout of the ECU and enables higher layers to access them in a uniform way. Memory services are responsible for managing non-volatile data, e.g., saving and loading data, error checking. Communication service provides the uniform interface and support required from the BSW layer for network communication, e.g., CAN, LIN. System services provides the collection of services used by modules of all layers, e.g., timer, error manager, diagnostic event manager, watchdog manager.

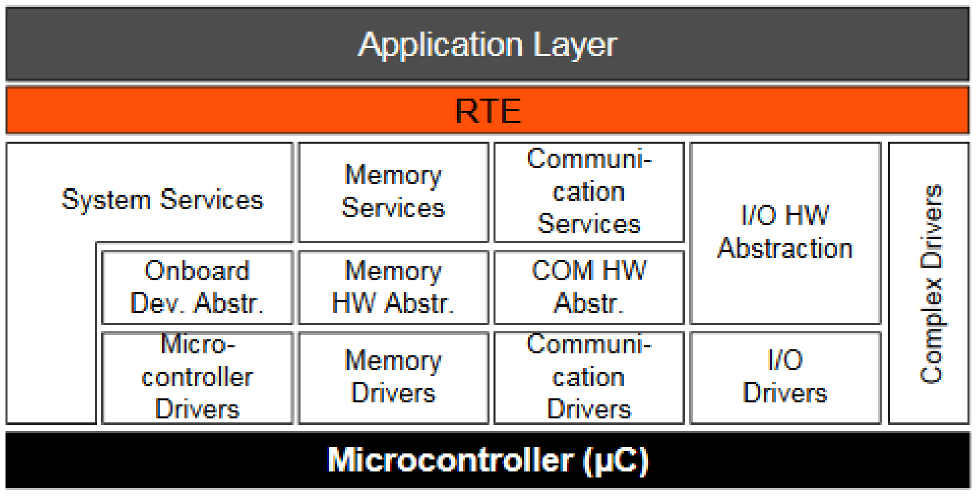


Figure 2.3: Detailed AUTOSAR architecture [5]

2.1.2.2 RTE Layer

RTE layer sits between the application layer and the BSW layer [9]. An AUTOSAR application access all the services through RTE layer. This makes AUTOSAR compliant software applications independent of the underlying infrastructure. Since needs of software applications vary application by application, RTE needs to be tailored based on the Software Applications hosted on an ECU. E.g., RTE generated for software application having software components communicating with other components on the same ECU or with components running on another ECU will require different communication support from RTE. For the later inter-ECU communication case, RTE will have to support additional communication channels, e.g. CAN, LIN. This application specific tailoring makes the RTE generated for one ECU different from another ECU.

2.1.2.3 Application Layer

The application layer is the topmost layer of the stack and is not standardized. All the automotive software applications run in this layer. In the application layer, architectural style is not layered but component based. All the functionalities are implemented using software components which can use infrastructure through standardized interfaces. A Software Component (SWC) is the smallest part of a software application that has a specific function [10]. It is a fundamental design concept which separates the infrastructure from the application. An application's composition is hierarchical and consists of one or more software components. There are two types of software components:

1. **Atomic software component**: A simple SWC running on only one ECU is called atomic SWC.
2. **Composed Software Components**: A SWC that is composed of more than one atomic or composite SWC and runs on more than one ECU is called composed SWC.

The functionality inside a SWC is implemented using Runnables. A runnable is the smallest code fragment of an SWC. It is very similar to a function in standard programming languages. Runnables implement only the application logic, for the system level services they call RTE layer APIs. A runnable has to be mapped to an OS task to get scheduled. OS tasks are the schedulable entities of the AUTOSAR OS which are similar to Linux processes.

Software components communicate and access system services using two types of ports:

1. **PPort**: Ports which are used to provide data as defined in the application interface.
2. **RPort**: Ports which are used to receive data as defined in the application interface.

There are two types of communication supported by the AUTOSAR standard:

1. **Sender-Receiver**: In sender-receive communication, a sender can send information to one or more receivers and vice versa. The sender and receiver are completely decoupled and are unaware of each other's location. This communication can be queued or un-queued. Furthermore, sender-receive communication can be divided into two different types – **Implicit** and **Explicit** sender-receiver communication. In implicit communication, all the data read by a runnable is copied to its local environment before starting its execution and all the values written by it are written from its local environment after its execution is finished. In explicit communication, contrary to implicit communication, all the data is written and read, to and from the ports, instantly.
2. **Client-Server**: In client-server communication, client requests the data from the server. A client can invoke many servers and vice versa. Cient-server communication also can be divided into two categories – **Synchronized** and **Asynschronized** client-server communication. In the Synchronized client-server communication, the client invokes the server and the server executes within the same task using the resources of the client's task. Once the server is done, control returns back to the client and client resumes from the same point where it left. It is very similar to a nested function call. In Asynchronized Client-Server communication, the client invokes the server and proceed to the next instruction which means that client does not wait for the server to finish. Once server is done it raises an event for which client runnable has subscribed. The data can be passed as a message or using a shared buffer.

SWCs communicates with other SWCs on same or different ECUs using Virtual Function Bus. Virtual function bus is logical concept which provides the abstraction for communication services [11]. A SWC will only see VFB and not the hardware dependent code. For intra-ECU communication VFB is realized using RTE while for inter-ECU communication it is implemented by both RTE and BSW. **Figure 2.4** shows an example of intra and inter-ECU communication. SWC-B and SWC-C are hosted on the ECU-2 while SWC-A is hosted on ECU-1. The communication between SWC-B and SWC-C is implemented by the RTE layer of the AUTOSAR stack hosted on ECU-2. However, in order to communicate with SWC-A, SWC-B and SWC-C are using the communication mechanism implemented by both RTE and BSW layers of the AUTOSAR stack running on ECU-1 and ECU-2. Additionally, as shown by the SWC-C's communication with the sensor, access to hardware device requires support from entire AUTOSAR stack.

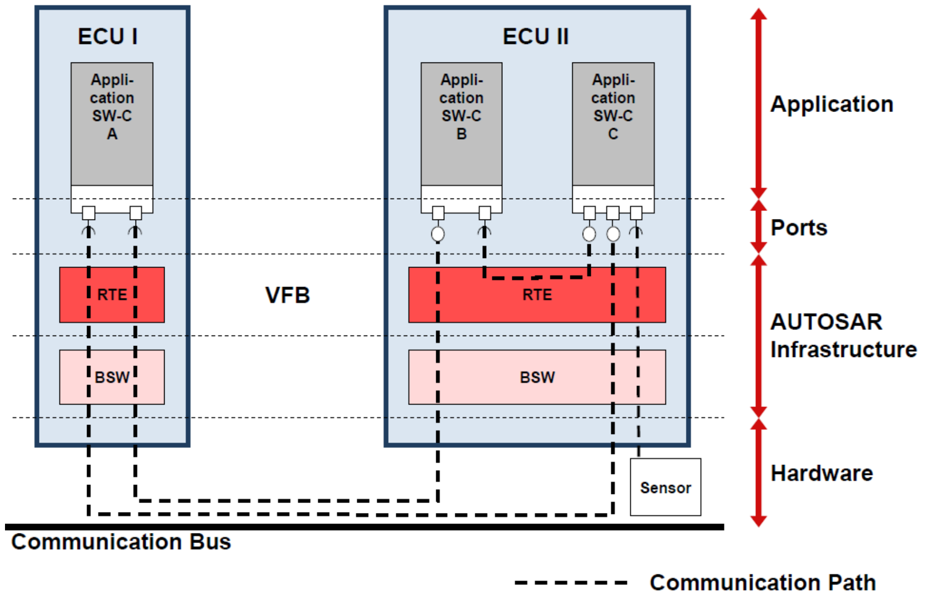


Figure 2.4: Intra and inter-ECU communication [5]

2.1.3 Operating System

AUTOSAR has specified a set of requirements for the operating system so that it complies with the rest of its architecture and can be used by all the vehicle domains. These requirements are specified in the document "Requirements in Operating System" [25]. Any OS, whether proprietary or open source, can be used in AUTOSAR stack if it supports all the requirements specified by the AUTOSAR standard. While drafting these specifications OSEK OS (ISO 17356-3) [8] served as the basis. The OSEK is a single processor OS that provides all the functionalities required to support event driven control systems. The priority of tasks is statically defined, and hence the user cannot change them at the time of execution. OSEK supports both preemptive and non-preemptive scheduling. Furthermore, OSEK supports two types of ISRs (Interrupt Service Routines) – ISR1 and ISR2. ISRs of category one do not use any OS service and hence have no effect on task management. ISRs of category two uses OS services in routine to handle the interrupt. Furthermore, OSEK uses immediate priority ceiling protocol to avoid priority inversions.

All the services expected from OS together with configuration are described in an XML file. As mentioned in **Section 2.1.4** in detail, based on the description in the OS configuration file, a code generating tool generates the OS code which is compiled with the rest of the code of the system to generate the executable for the ECU.

2.1.4 Methodology

AUTOSAR methodology defines the approach of building the ECU software. The whole process is devised to solve the problems targeted by AUTOSAR consortium. All the supplier and product developers can build their products independent of each other by following standard interfaces and using standard data exchange formats. **Figure 2.5** shows the basic steps of the methodology. As the first step, the vehicle manufacturer fixes the requirements of the functionality which need to be implemented in the vehicle. The next step involves finding different parts of the whole E/E architecture which satisfy its requirements best. Mainly, these parts are Software Application, software tools to configure and generate AUTOSAR stack, and the ECU. Next, following main standard interfaces are described:

- SWC descriptions
- ECU resources descriptions
- System descriptions

Software descriptions contain the architecture and interface for the SWC, e.g., runnable names, required services from the system, runnable communication details, and interfaces. ECU resource description contains the details about all the hardware resources available for the application, e.g., CPU, the number of DIO pins. System description contains all the system-wide information which might be spreading over many ECUs. All these documents are defined using XML. In the second step, the entire system is configured using software tools in accordance with the SWC, ECU resource, and System descriptions. For example, Arctic Studio, an AUTOSAR stack configuration tool by vendor Arctic, contains RTE and BSW editors in its sets of tools to configure the RTE and BSW layer for an AUTOSAR application [3]. As an output of this step, ECU descriptions are generated. An ECU description contains all the details about entire stack for a specific ECU. Next, each ECU is configured according to its description. This configuration includes all the details about how the application requirements are met by the application and AUTOSAR stack

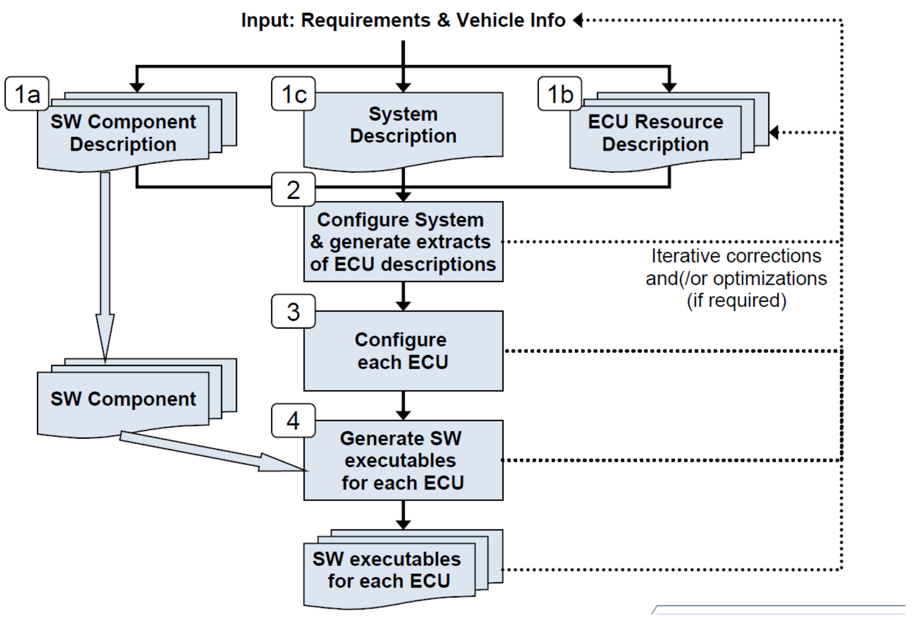

Input: Requirements & Vehicle Info
1a
SW Component Description
1c
System Description
1b
ECU Resource Description
2
Configure System & generate extracts of ECU descriptions
Iterative corrections and(/or optimizations (if required)
3
Configure each ECU
SW Component
4
Generate SW executables for each ECU
SW executables for each ECU

Figure 2.5: Overview of AUTOSAR Implementation Process [5]

on this ECU. In the fourth step, typically, software tools are used to generate all the RTE and configurable BSW layer code for a specific ECU. The details of this step are shown in **Figure 2.6**. Based on the configuration of each component of AUTOSAR stack, stack generating software tool generates the code that implements a component according to its configuration in ECU description. Generally, stack generating software tool consists of many tools, each of which is responsible for generating one component of the AUTOSAR stack, e.g., RTE generator, OS generator, MCAL generator, COM (Communication services) generator. This code generation is supported by the libraries implementing various functionalities, e.g., communication, transport protocols. Generally, these libraries are provided by the AUTOSAR stack vendors along with the tools to generate the AUTOSAR stack generating tools, e.g. Arctic Studio and Arctic Core [3][2]. Since the information in ECU configuration is tailored according to the SWCs requirements, the generated AUTOSAR stack is specific to an application and SWCs hosted on this ECU.

Next, for each ECU in the system, rest of the non-configurable code of the system and the implementation of SWC that are hosted on this ECU are added to the generated code, and a standard, microcontroller suitable compiler is used to compile this entire code. After successful compilation, an executable is generated for each ECU that implements all the expected functionality on a specific ECU. As the final step, the executables are run on their

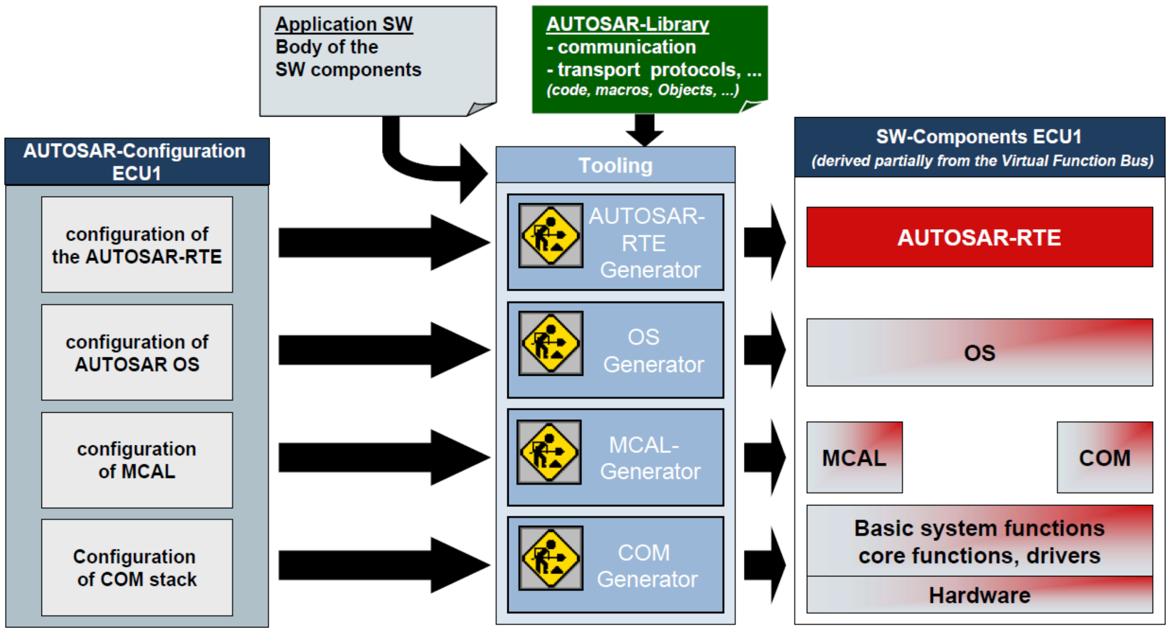

Application SW
Body of the SW components
AUTOSAR-Library
- communication
- transport protocols, ...
(code, macros, Objects, ...)
AUTOSAR-Configuration ECU1
configuration of the AUTOSAR-RTE
configuration of AUTOSAR OS
configuration of MCAL
Configuration of COM stack
Tooling
AUTOSAR-RTE Generator
OS Generator
MCAL-Generator
COM Generator
SW-Components ECU1
(derived partially from the Virtual Function Bus)
AUTOSAR-RTE
OS
MCAL
COM
Basic system functions core functions, drivers
Hardware

Figure 2.6: Generating the ECU executable [5]

respective ECUs to implement the application functionality. Among the steps marked in the **Figure 2.5**, steps 1a, 1b, and 1c can be executed in any sequence and by any vendor while steps 2,3, and 4 have to be followed in the order mentioned and for each of the ECUs. If required, the steps of this methodology can be iteratively repeated for corrections and optimizations discovered during the development process.

2.2 MAST

MAST (Modelling and Analysis Suite for Real Time Applications) is an open source set of tools to model and perform the timing analysis of an application [23]. MAST does operation-level characterization, allows specifying the timing constraints in its models and also performs schedulability analysis with them. It takes MAST model as an input which contains the description of a system using a set of pre-defined elements. MAST performs timing analysis on the model supplied as input using the technique selected by the user and then publishes the results to the user.

The tools MASTS provides are shown in **Figure 2.7** [20]. They are divided into the following categories:

1. **Design Tools**: These are the tools which convert models from other tool or formats to MAST Models, e.g., MARTE or UML to MAST [15].
2. **Data management**: These are the tools which are used to manage the input, output data or to convert them from one supported format to another, e.g., graphical result viewer, XML converter.
3. **Analysis Tools**: These are tools used in performing the timing analysis, e.g., schedulability analysis, sensitivity analysis.

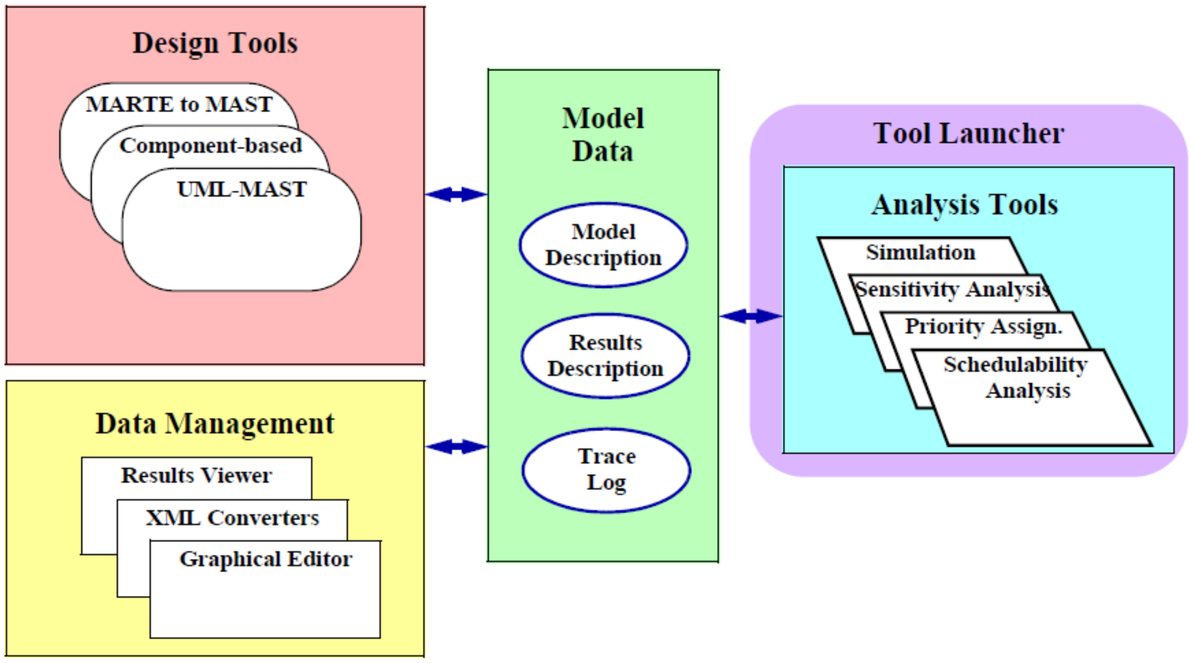


Figure 2.7: Tools provided by MAST [20]

2.2.1 MAST Model and its Elements

An application needs to be modeled as a MAST model to perform the analysis on it. For the MAST model descriptions, two formats are supported – text based special format, XML format. MAST models can be created manually or using a GUI-based editor, or translated from existing models using the tools provided by MAST. Some important elements of the MAST model and their usage is discussed below. Some of the self-explanatory attributes from the structures of the elements listed below are omitted.

2.2.1.1 Processing Resources

Processing resource is used to model the processing capacity of any hardware component. It can be of two types Regular_Processor and a Packet_Based_Network. A Regular_Processor has following main attributes in its structure:

```
Processing_Resource (
Worst_ISR_Switch => Normalized_Execution_Time,
Avg_ISR_Switch => Normalized_Execution_Time,
Best_ISR_Switch => Normalized_Execution_Time,
System_Timer => System_Timer,
...);
```

The attributes worst, average, and best ISR switches are used to represent worst, average, and best ISR switch overheads. System timer attribute has a reference to a concrete instance of the system timer element. It will be discussed in the following section. A Packet_Based_Network has following main attributes in its structure:

```
Processing_Resource (
Max_Packet_Transmission_Time => Normalized_Execution_Time,
Min_Packet_Transmission_Time => Normalized_Execution_Time,
List_of\_Drivers => ( Driver 1, Driver 2, ...),
...);
```

Max and min packet transmission times represent the maximum and minimum time taken in sending a packet over the network. These two are used to represent the packet transmission overheads. List of drivers attribute contains a list of network drivers representing the overhead models associated with the transmission of data over the network.

2.2.1.2 System Timers

System timer is used in modeling the way time events are handled by the system. It has attributes to represent the overhead associated with the timed event handling, e.g., worst, average or best case overheads. System timer can be of two types, Alarm Clock, and Ticker. The main difference between an alarm clock and a ticker is that while former raises an interrupt when an alarm expires, the latter raises the interrupt periodically and the system checks for the alarms that have expired during this period, if any. A system timer element has following structure:

```
System_Timer = (
Type => Ticker
Worst_Overhead => Normalized_Execution_Time ,
Avg_Overhead => Normalized_Execution_Time ,
Best_Overhead => Normalized_Execution_Time ,
Period => Time)
```

The attribute Period is ticker type system timer's additional attribute that is used to represent the time period of the periodic ticks.

2.2.1.3 Network Drivers:

Network drivers are used in modeling the operations executed as a consequence of sending or receiving any message over the network. There are three types of network drivers:

1. **Packet Driver:** This type of network driver is used to denote the drivers that get executed on transmission or reception of each message.
2. **Character Packet Driver:** It is a specialized version of packet driver used to represent the additional overhead associated with sending each character of the message, e.g. serial lines.
3. **RT-EP Packet Driver:** It is a specialized version of packet driver used to represent the characteristics of RTEP (Real-Time Ethernet Protocol).

Then network driver element has the following structure:

```
Driver = (
Type => Packet_Driver ,
Packet_Server => Scheduling_Server | Identifier ,
Packet_Send_Operation => Operation | Identifier ,
Packet_Receive_Operation => Operation | Identifier ,
...);
```

Packet server has a reference to a concrete instance of scheduling server that will execute this driver. Packet send and receive operations are the references to the concrete instances of the Operation element which need to be executed each time a packet is sent or received, respectively.

2.2.1.4 Schedulers:

Schedulers are used to model the operating system objects which implement the suitable scheduling strategies to manage the processing power they have been assigned. The structure of a scheduler element is following:

```
Scheduler (
Type => Primary_Scheduler ,
Policy => Scheduling_Policy ,
Host => Identifier);
```

The Scheduler could be of two types Primary Scheduler and Secondary Scheduler. Host attribute of primary scheduler has a reference to a processing resource whose processing power this Scheduler will distribute between the tasks which are assigned to it. The secondary scheduler has an attribute Server instead of the Host, which has a reference to a scheduling server. In other words, a primary scheduler can schedule servers or tasks and a server can schedule other servers or tasks through the secondary scheduler. The Scheduling Policy has a reference to a concrete instance of Scheduling Policy element that defines the scheduling strategy for this scheduler. **Figure** 2.8 shows the hierarchical structure of the scheduler element that is described here.

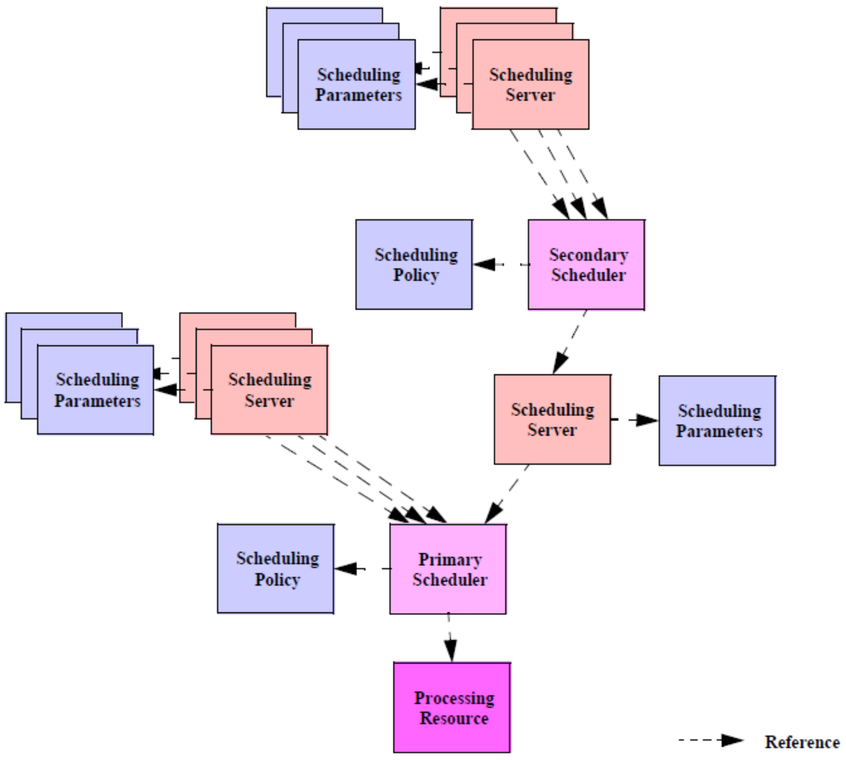


Figure 2.8: Hierarchy of scheduler and its associated elements[20]

2.2.1.5 Scheduling Policies:

The scheduling policy element defines the strategy that is used by a scheduler to deliver the processing power it has been assigned. Scheduling policy has the attributes to represent the best, average and worst context switch overheads. Its type field can be used to select one of the MAST supported scheduling algorithm, e.g., fixed priority, EDF (Earliest Deadline First). The structure of the scheduling policy element is following:

```
Scheduling_Policy (
Type => Fixed_Priority ,
Worst_Context_Switch => Normalized_Execution_Time ,
Avg_Context_Switch => Normalized_Execution_Time ,
Best_Context_Switch => Normalized_Execution_Time ,
...);
```

2.2.1.6 Scheduling Parameters:

Scheduling parameters contains the information required by a scheduler to make scheduling decisions about the associated scheduling server. The structure of a scheduling parameters element is following:

```
Sched_Parameters = (
The_Priority => 18,
...);
```

The type attribute is used to represent scheduling policy, e.g. fixed priority policy, non-preemptible fixed priority policy. The Priority attribute is used to represent the priority of the associated task.

2.2.1.7 Scheduling Servers:

Scheduling server element is used to model the schedulable entities, e.g., processes, threads. The structure of a scheduling server is following:

```
Scheduling_Server (
Server_Sched_Parameters => Sched_Parameters,
Synchronization_Parameters => Synch_Parameters,
Scheduler => Identifier ,
...);
```

The server scheduling parameters, synchronization parameter and scheduler attributes of scheduling server contain a reference to a concrete instance of their respective elements.

2.2.1.8 Shared Resources:

This element is used to model the shared resources which should be accessed in the mutually exclusive. The structure of shared resource element is as follows:

```
Shared_Resource (
Type => Immediate_Ceiling_Resource ,
Ceiling => 80,
...);
```

The type attribute is used to specify the protocol used to manage the shared resource accesses. The protocols supported by MAST are immediate ceiling protocol, priority inheritance protocol, and stack resource protocol [18]. The ceiling attribute represents the ceiling priority of the shared resource that is the priority of the highest priority task among all the tasks accessing this resource.

2.2.1.9 Operations:

Operations are used to model any executable piece of the code, e.g., function, message. The operation element can be of a simple or composite type. The Simple type operation has the following attributes,

```
Operation (
Type => Simple ,
Worst_Case_Execution_Time => Normalized_Execution_Time ,
Avg_Case_Execution_Time => Normalized_Execution_Time ,
Best_Case_Execution_Time => Normalized_Execution_Time ,
Shared_Resources_To_Lock => ( Identifier , Identifier , ...),
Shared_Resources_To_Unlock => (( Identifier , Identifier , ...),
...);
```

The Worst, Average, and Best case execution times attributes represent the worst, average and best case execution time, respectively. If an operation accesses shared resources mutually exclusively then the Shared Resources List attribute of the operation can be used to list all the accessed shared resources. The Composite type operation element has the following attributes:

```
Operation (
Type => Composite ,
Composite_Operation_List => (Identifier , Identifier , ...)
...);
```

The Composite Operation List attribute is used to list all the simple or composite Operation which a composite operation is comprised of.

2.2.1.10 Events:

Events elements are used to model the events occurring in the system, e.g., timer expired, message received. Events are categorized into two categories, Internal and External. Internal events are the events generated by an event handler. On the other hand, external events are the events that are not generated by event handlers. Event handlers are the actions that need to be performed when one or more events occur. Event handlers are discussed in coming sections in detail. Internal event elements have an attribute Timing Requirements that refer to a concrete instance of the timing requirements element. Following is the structure of an internal event:

```
Internal_Event = (
Type => Regular ,
Timing_Requirements => Timing_Requirement ,
..)
```

An external event can be of following types:

- **Regular:** A general event
- **Periodic:** An event that needs to be generated periodically
- **Singular:** An event that will be generated only once at the absolute time defined by its additional attribute Phase
- **Sporadic:** An aperiodic event that will have minimum inter-arrival time as specified by its additional attribute Min Interarrival
- **Burst** An aperiodic event that can occur in burst; it has additional attributes to describe the number of events and their distribution in a given time period

The structure of external event of Periodic type is presented below. The Period attribute specifies the period for this periodic type external event.

```
External_Event = (
Type => Periodic ,
Period => Time,
...);
```

2.2.1.11 Timing Requirements:

This element is used to model the constraints applicable to an associated event, e.g., timing requirements of generation of an associated event. The structure of timing requirement element is following:

```
Timing_Requirement = (
Type => Hard_Global_Deadline ,
Deadline => Time,
Referenced_Event => Identifier)
```

Deadline attribute represents the time limit imposed by the constraint. The type attribute defines the restriction on the generation of the associated event. The type could be hard-local, hard-global, soft-local or soft-global. In the case of a local type timing requirement, the deadline is imposed on the duration when the event activating the associated event generating runnable arrives till the associated event is generated. On the other hand, for global type timing requirement, one additional attribute Referenced Event is defined and the deadline is imposed on the duration from the referenced event occurred till the associated event is generated.

2.2.1.12 Event Handlers:

These elements are used to model the actions triggered by the arrival of an event. The structure of event handler element is following:

```
Event_Handler = (
Type => Activity ,
Input_Event => Identifier ,
Output_Event => Identifier ,
Activity_Operation => Identifier ,
Activity_Server => Identifier)
```

Type attribute represents the type of the event handler, e.g. Activity, System Timed Activity. Activity is used to represent the instance of an associated operation to be executed by an instance of scheduling server. These instances of operation and scheduling server are referred by activity operation and server elements, respectively. The event that starts this activity and the event that gets generated by this activity can be represented using the attributes input and output events. Input and output event attributes have references to the concrete instances of internal and external event elements, respectively.

2.2.1.13 Transactions:

This element is used to model the interrelated event and their respective handlers in the system. The structure of this element is as follows:

```
Transaction (
Type => Regular ,
Event_Handlers => (Event_Handler 1, Event_Handler 2, ...)
External_Events => (External_Event 1, External_Event 2, ...),
Internal_Events => (Internal_Event 1, Internal_Event 2, ...),
...);
```

Event handlers, external event, internal event, contains the list of concrete instances of the event handlers and their associated external and internal events, respectively.

Figure 2.9 shows the association between various MAST elements as described above.

2.2.2 MAST Results

After running the chosen analysis technique, MAST outputs a result file. This result file contains predefined system performance parameters and their values. These values are

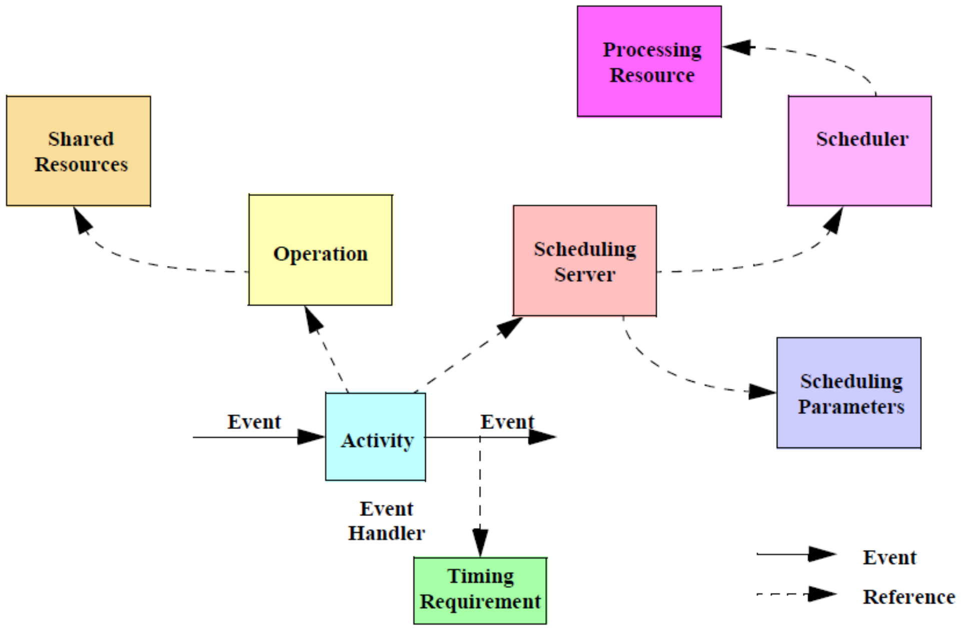


Figure 2.9: Different elements of a MAST model and their relations[20]

the performance metrics which estimate how well the system performed in the mentioned environment under the specified constraints. Slack is the percentage by which the execution time of an associated element can be increased while keeping the system schedulable, e.g., system slack, transaction slack, operation slack. Some of the parameters a result file can have are following:

- overall system slack
- processing resource specific slack and utilization
- transaction specific slack and best, average and worst response times for all of its events
- operation specific slack
- shared resource specific queue sizes

Chapter 3

Proposed Method

Testing and verifying the performance of any system in every stage of its development is very important for avoiding the repetition of work. This approach helps in reducing the development time and the cost of the product. In this respect, the design phase of an AUTOSAR application development process is critical, mainly because:

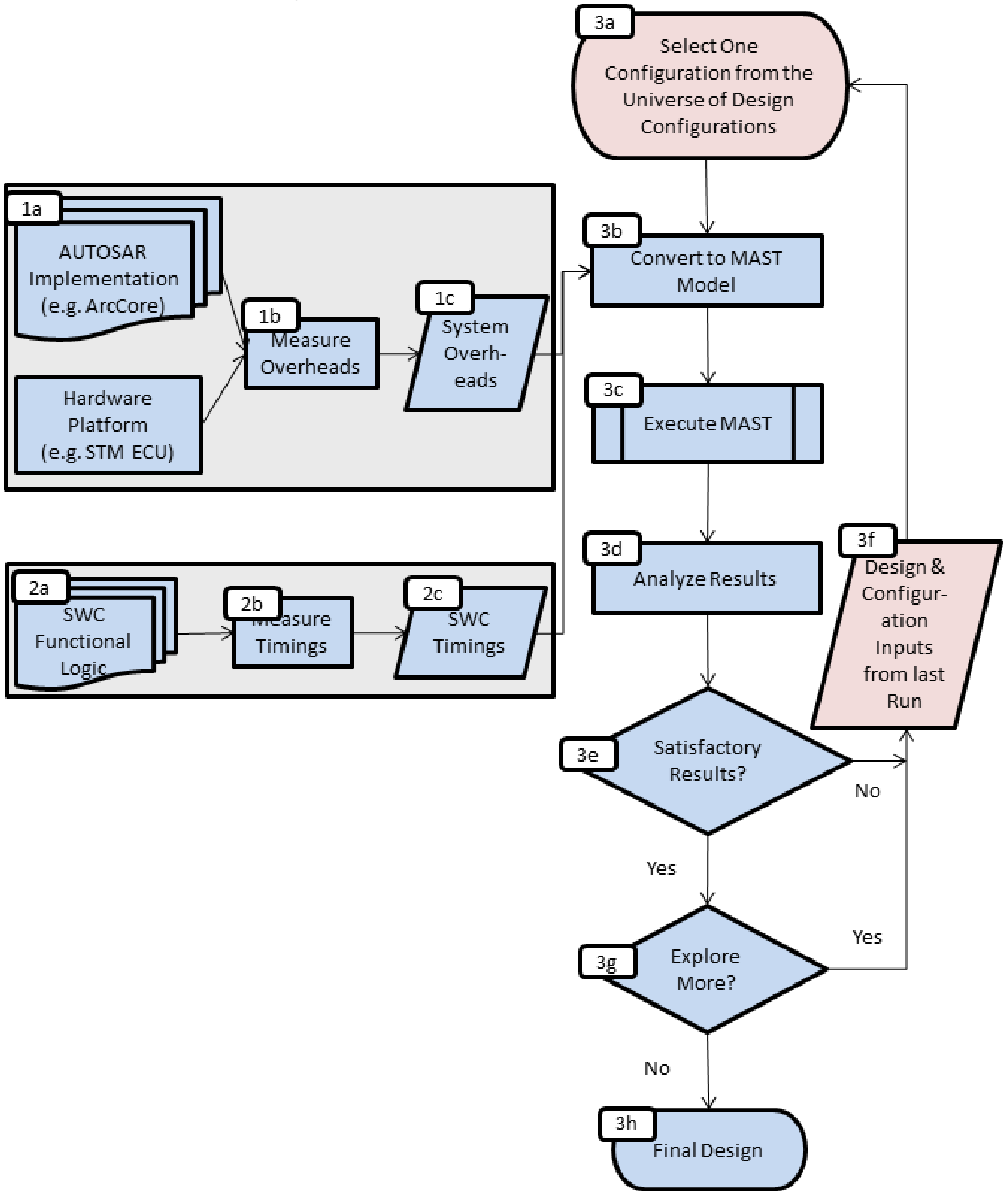
1. In the design phase of application development one has almost all the information about the application that is required for effective performance estimation.
2. The design of the application is the first concrete step in its development. Any design error could be very costly, with effects from increased cost to the total failure of the project.

AUTOSAR applications function in a safety critical environment. If they do not perform as intended then the effects could be fatal. Thus, AUTOSAR applications have stringent safety requirements which have to be met under all circumstances. These safety requirements make the schedulability analysis of the AUTOSAR applications even more important. Unfavorably, the complex nature of the AUTOSAR applications makes the procedures and tests to verify their schedulability error prone. One important factor to consider during the performance analysis is the overhead added by the AUTOSAR stack. Everything, apart from the application, executed by the CPU is considered overhead, e.g., context switch, communication overhead, calls to any RTE API, timer management. Generally, these overheads are either ignored completely or considered not to have a great impact on the performance and schedulability of the AUTOSAR application. This fact reduces the precision of the estimated performance, and thus, compromises the designer's reliability on performance estimations. A precise estimation of the performance in the application's design phase can give designer leverage to efficiently explore possible alternative designs and configurations of the system. To get a factual application performance estimate and thus more reliable application design, it is essential to consider the system overheads in the schedulability analysis of the application. This chapter proposes a method to address this problem. The proposed method tests and verifies the schedulability and performance of the AUTOSAR application while considering all the overheads added by the AUTOSAR stack. This method can also be used to select the hardware, AUTOSAR stack vendors, and other resources needed for deployment or migration from one system to another.

3.1 Overview

The goal of this method is to find a design configuration for an AUTOSAR application that will make it schedulable. Once the AUTOSAR application designer has enough information to calculate application execution times, the next step is to select the tools to generate an AUTOSAR stack for this application, and the ECU which will host this application. Even though all the tools which generate AUTOSAR stack abide by the same standard, their performance can still differ based on the quality of their implementation of the AUTOSAR standard. This means that different tools will generate different AUTOSAR stacks, which will thus have different overheads. The ECU needs to be selected based on the requirements of the application under development and the budget available. For this ECU, the next step is to measure the application running time and the overheads of the stack generated by the selected tool. After these measurements, one design configuration needs to be selected by the designer based on the available knowledge about the system and his own expertise. This configuration will be translated to the MAST model with application running time and the system overheads. In the next step, MAST will be executed on this model. If the performance is not satisfactory or the designer wants to explore more configurations, the MAST output can be used as input to decide on the next configuration. Otherwise, this configuration is selected as final and can be followed in the rest of the application development. **Figure 3.1** shows all the steps of the proposed method. These steps are discussed in detail in the following sections.

Figure 3.1: Steps of the proposed method



3.1.1 Step 1: Measuring the AUTOSAR Stack Overheads

Ideally, the vendors providing the tools for generating the AUTOSAR stack should also provide the overheads matrices for each of the overheads for their AUTOSAR stack implementation for the supported ECUs. The application designer will thus not have to measure them. Nevertheless, if one has to measure the overheads of an AUTOSAR stack for a given ECU, there are many ways of measuring the execution time of a piece of code which can be used to measure the overheads, e.g., using the system timers. Accessing the system timer and using it for execution time measurement requires understanding the system timer related code. However, in the case of AUTOSAR application, the AUTOSAR stack code is supplied by a different vendor. Understanding this code and modifying it to measure the overheads could be cumbersome for the application designer. In such cases, an external timer can be used to measure the overheads. This is the method used in the evaluation of the proposed method and is described in the **Section** 4.2.3.

The output of this step is the different types of overheads added by an AUTOSAR implementation running on a specific ECU. These overheads can be different if the AUTOSAR stack or the ECU is changed. Furthermore, these overheads may or may not be AUTOSAR application dependent, e.g., overhead added by PWM-Write RTE API is an application independent while communication overhead is application dependent since it depends on the several other factors listed in **Section 4.3.1**. For the application dependent AUTOSAR stack overheads, vendors can provide the overhead measurements as input dependent matrices so that the application designers will not have to measure them. In the evaluation of this method, one such matrix is provided for the communication overhead in **Section 4.3.1**. This matrix outputs the time taken by sending a message for a message size range from 1B to 4KB. This matrix is specific to the explicit sender-receiver communication with no message queue and when the communicating runnables are mapped to the same task. As listed in the **Section 3.2.2**, in addition to the message size, communication overhead depends on 4 types of communication, 3 runnable mappings, and 2 communication configurations. As a result, AUTOSAR stack vendor will have to provide 24 matrices (4x3x2) for each of the supported ECUs. Out of these 24 matrices, each matrix will take the message size as input and will provide the communication overhead for one of the possible 24 cases.

3.1.2 Step 2: Measuring SWC Timings

An AUTOSAR application is composed of many SWCs, and these SWCs are implemented using runnables. The execution time for each of these runnables needs to be measured.

The time measurement should cover all the functionality's parts and should not include RTI API calls or any other AUTOSAR stack code fragment. RTE APIs and system service time execution is excluded from the SWC time measurement because its application independent. These APIs takes constant time for an AUTOSAR stack running on a specific ECU. Measuring them as AUTOSAR stack overheads reduces the trouble of measuring them for each application. Additionally, since all system overheads are measure separately, one doesn't have to complete the implementation of AUTOSAR application and generate AUTOSAR stack code for it in order to measure the execution time of application.

For this time measurement, a static code analysis tool can be used. Another way of capturing these execution times is by using an external timer as mentioned in the method described in **Section** 4.2.3. The output of this step is the execution time of different SWCs of the AUTOSAR application.

3.1.3 Steps 3: MAST Execution and Result Analysis

Once the AUTOSAR stack overheads and application execution time are measured, the next step is to model the AUTOSAR application into a MAST model. Later, MAST is executed on this model. Depending upon the requirement one can select one of the many analysis tools provided by MAST, e.g., Offset_Based, EDF, Holistic. If there are no precedence constraints between different tasks, i.e. no transaction across multiple tasks, then Classic RM analysis technique can be used. Otherwise, offset-based analysis technique can be used.

As described in **Section** 2.2.2, this execution will generate a result file. Using the GUI of MAST Analyzer, the application designer can set the parameter he wants in the result file. The application designer has to analyze this file and check whether the application in the current environment is schedulable and meets the performance goals. If not, then using the inputs from these results and his own expertise, the application designer can decide the next design configuration to repeat the steps marked as "3" in Figure 3.1. Also, a designer might want to explore more configurations, even after getting a schedulable design configuration. In such cases, the process described in step 3 needs to be iteratively repeated. Steps marked as "1" and "2" need not be repeated in this iterative process. Furthermore, steps marked with "1" will be repeated only if either the hardware being used or the AUTOSAR stack is changed; steps marked with "2" will need to be repeated only if the application's functional logic is changed.

This thesis focuses primarily on the quantification of AUTOSAR stack overheads and the process of modeling an AUTOSAR application into MAST.

3.2 Quantifying AUTOSAR Stack Overheads

As mentioned in **Section 2.1** some part of AUTOSAR stack is generated specifically for an application. It is tailored according to the application requirements. Later, this generated stack is compiled with the AUTOSAR application implementation to generate the executable, which will be run on an ECU to implement the required functionality in the vehicle. While there are many advantages of using AUTOSAR, there is also a cost to it – overhead. A timing overhead is the time taken by the CPU to execute any code that is not a part of the application. In other words, everything that needs to be executed by the CPU is considered overhead, except the application code. This overhead mainly comes from various services used by the application or the management done to run the application, e.g., process context switch, IO. An application is divided into SWCs, which are implemented through runnables. Following code snippet presents an example of a runnable:

```
/* Runnbale implementation */
void swcHeadlightMainRunnable(void){
  SignalQuality quality = 0;
  DutyCycle dutyCycle = 0;
  float64 headLightState = 0.0;
  Std_ReturnType ret = E_NOT_OK;

  ret = Rte_Read_lightBrightnessReceiver_lightBrightness( &headLightState);

  if(RTE_E_OK != ret){
    /**** error handling code ****/;
  }

  if(BEAM_HIGH == headLightState)
    headLightState = VOLTAGE_MAX;
  else if(BEAM_LOW == headLightState)
    headLightState = (VOLTAGE_RANGE/2.0);
  else
    headLightState = VOLTAGE_MIN;

  ServoMotorControllerFunction(&headLightState, &dutyCycle); /* local
   function call */

  Rte_Call_HeadlightPwmDuty_Set(dutyCycle, &quality);

  if(RTE_E_PwmServiceSetDuty_E_OK != ret){
    /**** error handling code ****/;
  }
}
```

Runnables are not schedulable entities but rather a piece of related code that join together to implements a part of the functionality of an SWC, e.g., swcHeadlightMainRunnable mentioned above is a runnable of SWC Headlight and is responsible for controlling the headlight. This runnable executes its code and then communicates using Rte_Read_lightBrightnessReceiver_lightBrightness (..) RTE API. Based on the value received in the message, the runnable calculates the brightness and the state for the headlight of the car. It controls the head light using and Rte_Call_HeadlightPwmDuty_Set(..) RTE API.

A runnable is mapped to a task and gets executed when the event which activates this runnable has occurred and its task is scheduled. This runnable-to-task mapping is done in the RTE editor. The runnable swcHeadlightMainRunnable shown above was mapped to task OsHTask. The following code snippet shows the RTE code generated for this runnable:

```
    /* RTE code generated to invoke runnable */
    void Rte_swcHeadlight_SwcHeadlightMainRunnable(void) {
        /**** Pre-runnable execution processing ****/;
        swcHeadlightMainRunnable();
        /**** Post-runnable execution processing ****/;
    }
```

The following is a code snippet showing the code generated for the task OsHTask. In this task, runnable swcHeadlightMainRunnable will get executed when the event EVENT_MASK_OSEventStartH occurs.

```
/* Task to which runnable was mapped using BSW editor */
void OsHTask(void) {
    EventMaskType Event;
    do {
        SYS_CALL_WaitEvent (EVENT_MASK_OSEventStartH);
        SYS_CALL_GetEvent(TASK_ID_OsHTask, &Event);

        if (Event & EVENT_MASK_OSEventStartH) {
            SYS_CALL_ClearEvent(EVENT_MASK_OSEventStartH);
            Rte_swcHeadlight_SwcHeadlightMainRunnable();
        }

    } while (RTE_EXTENDED_TASK_LOOP_CONDITION);
}
```

OsHTask is a periodic task. It has registered an periodic alarm that will generate an event on its expiry. OsHTask waits for this event to occur. Whenever this event occurs, OsHTask's state is changed from waiting to ready. If this task is the highest priority task among all the ready and running tasks, then it will be scheduled for execution. In the code snippet for the task OsHTask, EVENT_MASK_OSEventStartH is the event that will be generated on the alarm expiry. SYS_CALL_WaitEvent(..) is the system call used by the OsHTask's to wait for this event to occur. Once the EVENT_MASK_OSEventStartH has occurred, OsHTask's execution will start. It will check which runnable has to be started on the occurrence of this event (event to runnable mapping is also done in the RTE editor). OsHTask clears the event and then calls the RTE function (Rte_swcHeadlight_SwcHeadlightMainRunnable (..)) to start this runnable. This RTE function will do some preprocessing required to start the runnable and then it will start the runnable. Now, the first instruction of swcHeadlightMainRunnable will be executed. Once the runnable's execution is finished control will go back to RTE layer function which started the execution of this runnable. This function will do some post-runnable processing and then control will go back to the OsHTask. Finally, since task does not have any other runnable mapped to this event, it will go back to the waiting state and will be activated periodically, each time this event occurs. The pre and post processing done by the RTE function mainly involves setting up the parameters as per the requirements of the services used by the runnable. For example, in case of implicit communication, preprocessing involves copying the data from the shared buffer to the variables accessible by the runnable. Similarly, postprocessing involves writing data from runnable accessible variables to the shared buffer.

In the last example of swcHeadlightMainRunnable, not everything executed by the processor was part of the runnable, e.g. RTE code to start the swcHeadlightMainRunnable. Execution of all the code that is not part of the runnable causes overhead. The following sections present the details about the different type of overheads added by the AUTOSAR stack.

3.2.1 OS Overheads

OS overhead is the overhead added by the AUTOSAR OS. Since OS is a part of AUTOSAR stack, OS overhead does not depend on the AUTOSAR application. The main types of overheads added by the OS are due to context switching, interrupt handling, and the system timer.

Context switching is a procedure that is followed when a task under CPU execution is switched. It can be initiated when a higher priority task becomes ready and preemption is allowed, or when the execution of the current high priority task is finished. The cost of context switching depends on the operations performed during the context switching. In general, the scheduler determines which task needs to be executed next. After that, dispatcher stores the contextual information of the current task under execution and loads the contextual information of the task that will be executed next. This contextual information commonly consists of register and memory maps, stack pointer and program counter. The time taken by the CPU in executing these operations is called a context switch overhead. In general, the time taken by the dispatcher is constant, however, the time taken by the scheduler depends on the number of tasks in the system. Due to this, the context switch overhead might vary based on the number of tasks used in an application. The context switch overhead can be measured by using the following method. The lower priority task should run infinitely. This can be achieved by putting an “infinite while loop” in the task. Higher priority task can be made periodic so that it will run each time after a fixed duration. The expiry of periodic alarm will make the higher priority task ready. This, in turn, will prompt the scheduler to assign the CPU to this task. Since the lower priority task runs infinitely, each time scheduler assigns the CPU to the higher priority task a context switch will happen. Similarly, after the higher priority task is finished, a context switch will happen again. Consequently, to measure the context switch, one simply has to measure the time when the execution of higher priority tasks is finished and the execution of the lower priority tasks is resumed.

The overheads associated with an interrupt handling can be divided into three parts interrupt entry, ISR execution time, and interrupt exit. Interrupt entry overhead is the time elapsed between an interrupt occurred and its ISR is called. Conversely, interrupt exit overhead is the time elapsed between ISR execution is finished and CPU resumes executing the highest priority task. These overheads amount to the cost of CPU execution done due to the occurrence of an interrupt. Since AUTOSAR OS has two types of overheads, the interrupt entry and exit time will be different for these two ISRs. Interrupt exit and entry time remain constant for an AUTOSAR stack implementation running on a specific ECU. However, these two overheads are independent of AUTOSAR application. The overhead caused by an ISR execution will vary from one ISR to another. As mentioned before, the ISRs of category two can use system services, so they have greater interrupt entry, ISR execution, and interrupt exit overheads. To measure the interrupt entry overhead for an AUTOSAR implementation, an interrupt can be generated based on an external event, e.g., analog input received over a pin on the hardware board. An ISR can be configured in the system to handle this interrupt. Now, to measure the interrupt entry overhead one simply has to measure the time difference between when the event that raised the interrupt occurred and when the first instruction of the ISR responsible for handling this interrupt was executed. ISR execution overhead for an interrupt can be measured by measuring the time taken in ISR execution. Finally, the interrupt exit time can be measured my measuring the time difference of the timestamps when ISR finishes and OS resumes to normal processing.

The system timer is the timer used by the OS for measuring the time to handle timed events. The AUTOSAR OS uses a periodic ticker to measure the time. After a fixed period, ticker raises an interrupt which signifies a tick. On interrupt arrival, OS checks for the counters which maintain the time in terms of the tick value. After each tick, OS reduces the counters by one. When a counter's value reaches zero, the associated alarm is raised which in turn generates the event associated with it. In the described process, all the instructions executed while receiving the tick interrupt periodically, maintaining counters, firing alarms, and generating events cause the system timer overhead. To measure the system timer overhead, one has to measure the execution time of the ISR that handles the tick interrupt. The system timer overhead will be the sum of this ISR execution time, and interrupt entry and exit times. The execution of timer ISR varies depending upon counters maintained and alarm it needs to fire when a tick occurs. As a result, the timer overhead will not be same for each tick. Designer has to consider the worst case execution time of the timer ISR when considering the system timer overhead.

Figure 3.2 shows different types of OS overheads for an example of two periodic tasks. The priority of the Task_2 is greater than the Task_1. Initially, Task_1 was being executed by the CPU and Task_2 is waiting for its period to expire. System timer overhead occurs each time a tick is received. An interrupt is raised by a hardware device during the Task_1 execution, which causes the interrupt entry overhead, its ISR execution overhead, and later interrupt exit overhead. After the interrupt handling is finished, Task_1 resumes execution. Next, on the expiry of the periodic timer for the Task_2, its state changes from waiting to ready. As a result, a context switch is triggered causing the context switch overhead. After the context switch, Task_2 starts executing once it gets the CPU. For the sake of simplicity, all other overheads are omitted from this figure.

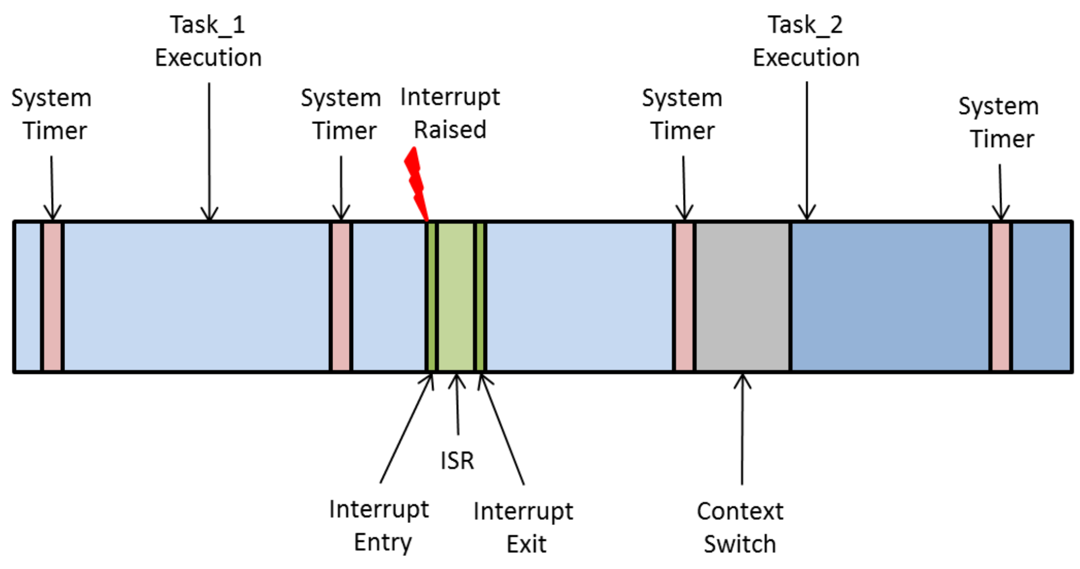


Figure 3.2: Different types of OS overheads

3.2.2 Communication Overheads

Different types of communication mechanisms are described in **Section 2.1.2.3**. In all types of communication mechanisms, messages are received or sent through RTE APIs, e.g., RTE API Rte_Read_lightBrightnessReceiver_lightBrightness(..) shown in the swc-HeadlightMainRunnable's code snippet. Calling the RTE APIs for communication invokes the execution of the code which is not a part of application logic. The CPU time taken in executing this code is called communication overhead. The implementation of the communication APIs can be different for different AUTOSAR stack implementations. Additionally, communication overhead depends on the following factors:

- **Communication type:** Implicit or explicit Sender-Receiver communication, Synchronous or Asynchronous Client-Server communication.
- Message size.
- **Communicating runnable's mappings to the tasks and their location:** Communicating runnables mapped to same task or two different tasks, running on the same ECU or two different ECUs.
- **Communication configuration:** Communication with or without message queue, server runnable with or without invocation request queue.

To measure the sender-receiver communication overhead, one can simply measure the execution time of the RTE API used to send or receive the message. This communication overhead is RTE API specific and will vary for different APIs depending on the factors listed above.

Measuring the communication overhead for client-server communication runnable is more complex. Synchronous client-server communication is blocking, i.e., calling runnable invokes the server runnable and waits for it to finish. The server runnable gets executed similar to a nested function call within the client runnable. Synchronous client-server communication overhead is measured in three parts. The first part is the time taken to invoke the server runnable. The second part is the time taken in executing the server runnable. The last part involves the time taken in returning the control from server runnable to the client runnable. The first and the third parts are comprised of the code generated by the RTE generator and hence remains same for an AUTOSAR stack implementation. The execution time of these two parts is application independent, but varies with the size of the data being communicated. This overhead can be measured irrespective of the application if the size of the data being communicated is known. Asynchronous client-server communication is non-blocking. The client invokes the server, but does not wait for the server to finish. However, the client has registered for an event which will get generated when the server finishes its execution. As a result, the client doesn't wait for the server, but still gets activated when the asynchronously invoked server is finished processing the request. For asynchronous client-server communication, the communication overhead can be measured by measuring the execution time of the RTE API. This overhead is specific to the application.

Network overhead is the overhead caused by sending and receiving packets during inter ECU communication. A runnable uses RTE APIs to sends or receives a packet over the network. Network overhead is caused by time taken in executing these RTE APIs. In addition to all the factor which affect communication overhead, network overheads can be affected by network topology, netwro capacity, network load, hardware used in the network, communication protocol. Network overheads are not covered in this thesis and will be targeted in the future.

3.2.3 Service Overheads

An AUTOSAR application accesses all the system services and resources through RTE APIs. E.g., in the presented code snippet, swcHeadlightMainRunnable is setting the brightness of headlight using the RTE API Rte_Call_HeadlightPwmDuty_Set(..). This API is generated as per the descriptions of this runnable and its configuration in the BSW editor. Initially, in the swcHeadlightMainRunnable's description, it is described that this runnable uses the PWM server of the AUTOSAR stack. In the BSW editor, all the details about its usage of the PWM server are configured, e.g., the channel to which PWM output will be generated. Moreover, in the BSW editor, the output of this PWM channel is configured as the input to the input channel of the headlight. All the code of AUTOSAR stack, including BSW and RTE is generated using this configuration. When swcHeadlightMainRunnable calls the Rte_Call_HeadlightPwmDuty_Set(..), it passes the value for the duty cycle. Since all the RTE and BSW code is tailored according to their respective configuration, they statically know all the details about the input received and output to be sent. In the example, the call to the RTE API will transfer the duty cycle value to the PWM server present in the BSW layer. In turn, PWM server will call lower layer functions to control the headlight. The CPU time used in executing all these functions is categorized as service overhead. All these service interfaces are standardized, but their internal implementation varies from one AUTOSAR stack implementation to another. This makes these overheads independent of the application, but specific to an AUTOSAR stack implementation. These overheads can be measured by measuring the execution time of their respective RTE APIs.

3.2.4 Runnable Overhead

Runnable overhead is the overhead involved in executing a runnable. It can be subdivided into following four parts:

1. Pre-Runnable-Event overhead

2. Post-Runnable-Event overhead

3. Pre-Runnable overhead
4. Post-Runnable overhead

Pre-Runnable-Event overhead is the time consumed in executing the code from the event received by the task to the RTE API responsible for starting the runnable execution is called. Pre-Runnable overhead is the time taken by this RTE API in starting the runnable execution since it is called. Post-Runnable overhead is the time taken since runnable execution is finished and control reaches back to task through the RTE APIs. Post-Runnable-Event overhead is the time taken by executing the task code from the RTE API responsible for starting the runnable till the task goes into the wait state through a wait system call. In the presented code snippet of the swcHeadlightMainRunnable, the pre-runnable-event overhead is caused due to the execution of the code from SYS_CALL_WaitEvent(..) function of the OsHTask till control reaches Rte_swcHeadlight_SwcHeadlightMainRunnable(..). Next, the overhead caused due to the code execution from this point till the first instruction of swcHeadlightMainRunnable(..) gets executed is called pre-runnable-event overhead. After this, the overhead caused due the code execution since the last instruction of swcHeadlight-MainRunnable(..) is executed till the control reaches back to the OsHTask (..) is called post-runnable overhead. Finally, all the OsHTask (..) code executed from here i.e. after Rte_swcHeadlight_SwcHeadlightMainRunnable() function call, till SYS_CALL_WaitEvent (..) system call causes the post-runnable-event overhead. The total runnable overhead for the swcHeadlightMainRunnable will be the sum of these four overheads. To calculate the runnable overhead, one has to measure the above listed four overheads. Individually, these overheads can be measured by measuring the execution time of the code that is causing them.

Runnable overhead could be different for different runnables. It depends on how much pre and post processing a mapped task and the RTE layer has to do before starting a runnable. E.g., for a runnable with implicit communication, since all the messages are read and sent before and after the runnable execution, the overhead will depend on message size and degree of communication of the runnable. This pre and post processing mainly involve setting the parameters as per the requirements of the services used by the runnable. It will differ based on the AUOSAR stack implementation. For the AUOSAR stack implementation used in the evaluation of this method, the only pre and post processing observed was to support the implicit communication of the runnable. **Figure 3.3** show different types of runnable overheads for an example using two runnables Ruannable_1 and Runaable_2. Both of these runnable are mapped to the same task and are activated by the same event. First runnable has no RTE API while second runnable has two explicit communication RTE APIs for receiving and sending two messages. One important thing to notice in the figure is that when the execution of the first runnable is over, post-runnable overhead is applied, however, post-runnable-event overhead is applicable only after execution of second runnable is finished. For the sake of simplicity, all the OS overheads are omitted from this figure. If the communication mechanism of second runnable is changed from explicit to implicit, its overheads will change. **Figure 3.4** shows the overheads for this case. Notice that the pre and post runnable overheads are increased because for an implicitly communicating runnable, all the messaged received and sent are read and sent before and after the runnable execution, respectively.

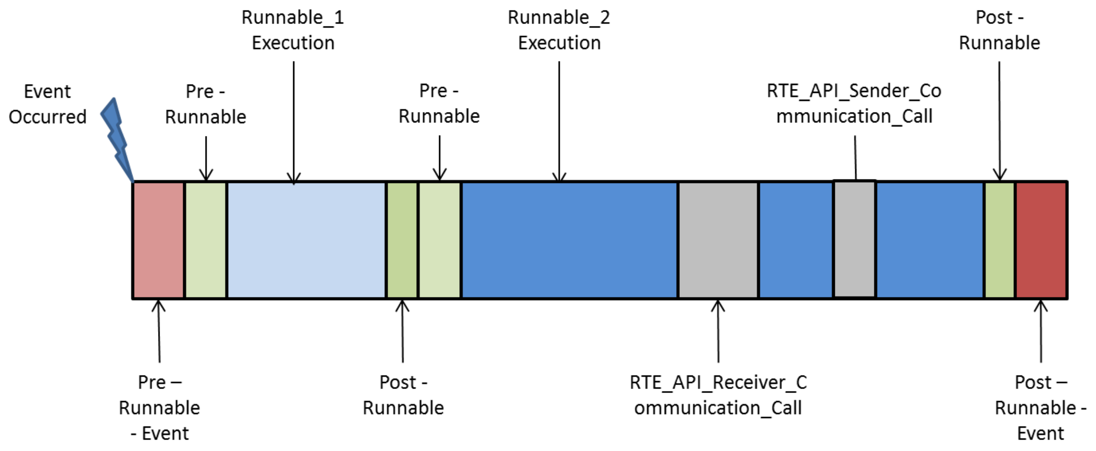


Figure 3.3: Runnable overhead for two runnables mapped to the same task and are being activated by same event. All communication done by Runnable_2 is explict communication.

One important point to remember during any overhead measurements is to measure the critical section separately. For example, if an RTE communication API uses shared buffers to implement communication between two runnbales mapped to two different tasks, then the access to shared buffer has to be mutually exclusive. AUTOSAR code generated for this RTE API will use a synchronization mechanism, e.g., a mutex, a lock or disabling interrupts, to guarantee that the access to shared buffer is always exclusive. In such cases, the time in executing code from RTE API to the critical section, critical section, and from critical section to the end of the RTE API has to be measured separately. This requirement comes from how this overhead is modeled into the MAST model of the affiliated AUTOSAR application which is explained in the next section.

In the beginning of the **Section 3.2** code snippets of swcHeadlightMainRunnable were presented and the control flow involved in the execution of this runnable was dis-

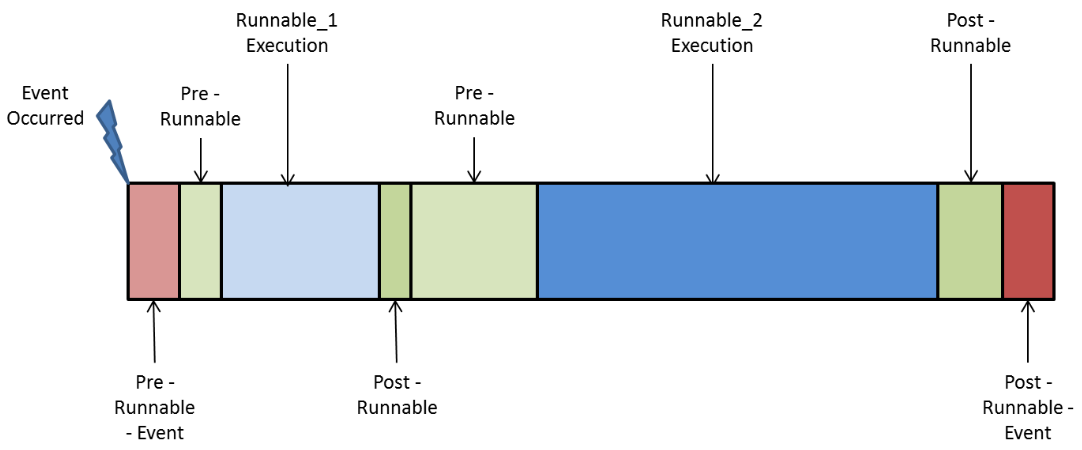


Figure 3.4: Runnable overhead for two runnables mapped to the same task and are being activated by same event. All coomunication done by Runnable_2 is implicit communication.

cussed. **Figure 3.5** represents the overheads involved in the execution of swcHeadlightMainRunnable. The first overhead is due to the system timer. Since a tick is periodic, the system timer overhead will be repetitive with the same period as of the tick. After the first tick interrupt received in the figure (marked as System Timer), the tick ISR will notice that counter responsible for firing the alarm that generates the event EVENT_MASK_OSEventStartH has reached the zero value. As a result, the event EVENT_MASK_OSEventStartH will be generated. A task in AUTOSAR waits for all the events of all of its runnables to occur. swcHeadlightMainRunnable(..) has subscribed for the event EVENT_MASK_OSEventStartH, so when this event will occur, the OS will change the state of OsHTask from waiting to ready. Now, once OsHTask becomes the highest priority task in the system, it will get scheduled for the execution. This will cause the context switch overhead. After getting the CPU, OsHTask will start its execution from SYS_CALL_GetEvent(..). The code from SYS_CALL_GetEvent(..) till Rte_swcHeadlight_SwcHeadlightMainRunnable() RTE API call will cause pre-runnable-event overhead. The execution of all the code from when this API is called and execution of the runnable swcHeadlightMainRunnable(..) begins will cause the pre-runnable overhead. Next, the swcHeadlightMainRunnable will start its execution. During the runnable's execution the function calls Rte_Read_lightBrightnessReceiver_lightBrightness(..) and Rte_Call_HeadlightPwmDuty_Set(..) will cause communication and service overheads, respectively. After the runnable execution is over, the execution of all the code from there till the control reaches OHTask(..) will cause post-runnable overhead. In the

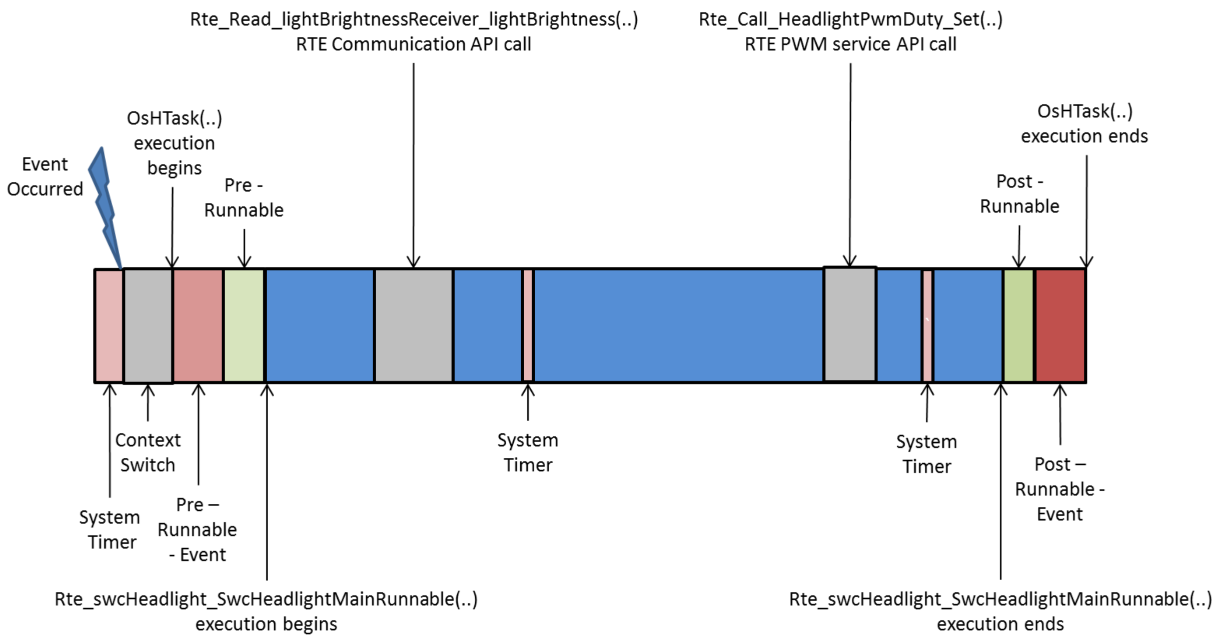

Rte_Read_lightBrightnessReceiver_lightBrightness(..)
RTE Communication API call
Rte_Call_HeadlightPwmDuty_Set(..)
RTE PWM service API call
OsHTask(..) execution begins
Event Occurred
Pre - Runnable
OsHTask(..) execution ends
Post - Runnable
Context Switch
System Timer
Pre – Runnable - Event
System Timer
System Timer
Post – Runnable - Event
Rte_swcHeadlight_SwcHeadlightMainRunnable(..) execution begins
Rte_swcHeadlight_SwcHeadlightMainRunnable(..) execution ends

Figure 3.5: Different types of overheads for swcHeadlightMainRunnable. The direction of execution is from left to right.

end, all the code executed from Rte_swcHeadlight_SwcHeadlightMainRunnable(..) till SYS_CALL_WaitEvent(..) function call of OsHTask will lead to the post-runnable-event overhead.

3.3 Creating a MAST model for an AUTOSAR Application

Implementing a functionality using AUTOSAR in a vehicle requires three things – hardware resources, AUTOSAR stack, and AUTOSAR application. **Section** 2.2.1 described various elements of a MAST model. By using these elements, hardware resources, AUTOSAR stack, and AUTOSAR application can be represented in a MAST model.

3.3.1 Modeling the ECU and Hardware Resources

All the hardware resources and their properties can be modeled using MAST elements. The available processor of an ECU can be represented using the Processing Resource element of MAST model. Each available computational resource of the ECU should be represented using a concrete instance of processing resources. A processing resource of Regular Processor type can be used to represent a physical core and its related attributes. The min and max interrupt priority attributes of the processing resource can be used to specify the minimum and maximum priorities of the interrupts modeled. It also has worst, average, and best ISR switch attributes that can be used to specify the worst, average and best overhead associated with the ISR switch. Furthermore, it has one system timer attribute that can be used to specify the system timer associated with this processing resource.

OSEK, the AUTOSAR OS, uses ticks to specify a counter for an alarm. A Ticker type System Timer element can be used to model the mechanism used to handle the timed events and its characteristics. The overheads associated with handling the alarms counter ticks, can be specified using the Worst, Average, and Best Overhead attributes of the ticker. The period of the ticker can be specified in the Period attribute.

Additionally, even though thesis doesn't focus on network, however, if required then the network that uses real-time protocols can be modeled using the Packet Based Network type processing resource, e.g., CAN bus. Packet based network element has attributes to represent various network characteristics, e.g., throughput, transmission type (Simplex/Half Duplex/Full Duplex), transmission time, packet size.

If the hardware resources have to be shared in a mutually exclusive way, then they can be represented using Shared Resource elements of MAST model. This element makes sure that the access to these resources is always mutually exclusive. Since, AUTOSAR OS uses immediate ceiling protocol for resource sharing and management, the Type attribute of the shared resource element should be set to Immediate Ceiling Resource. Additionally, since AUTOSAR OS computes the priorities of shared resource by itself, the Preassigned attribute should be set to No.

3.3.2 Modeling the AUTOSAR Stack

The AUTOSAR stack components that need to be modeled are the services used by the AUTOSAR application and the OS properties. BSW layer services are accessed by the application using RTE APIs. These APIs can be modeled using the Operation element of the MAST model. If an RTE API is divided into several parts for measuring the execution time, then each part has to be represented individually. In such cases, a Composite type Operation can be used to model the API. The Operation List of this composite operation will contain the references to simple or composite operations. Each operation referred in the list will represent the functions being used in the RTE API. The Time attributes of an operation element can be used to specify the worst, average and best execution times of its code.

A critical section can be represented by using the shared resource and operation elements. Each object used in ensuring the access to critical sections mutually exclusive can be represented using an instance of shared resources, and each critical section can be represented using a simple type operation. The shared resource list of this operation should list the shared resource protecting this critical section. Additionally, the time attributes of this operation can be used to represent the execution time of the represented critical section. For example, consider a code fragment with five critical sections. If one lock is used to guarantee the mutual exclusive access to these five critical sections, then the MAST model of it will have one shared resource and five simple operations. Each operation will list the same shared resource in their respective shared resource list. On the other hand, if one lock is used to guarantee the mutual exclusive access of each of the five critical sections, then there will be five shared resources in the model and the shared resource list of the each operation will contain its specific shared resource. The OS objects used in implementing the scheduling strategy can be represented using a Primary Scheduler element of the MAST model. Since priorities of the tasks in AUTOSAR are statically defined, the Policy attribute of scheduler should refer to an instance of Fixed Priority type Scheduling Policy. Moreover, policy attribute can also be used to represent the context switch attribute.

All the interrupts, ISRs and tasks running system services or OS functionalities should be modeled, e.g. networking service, BSW service, external interrupts and their respective ISRs. A Regular type Scheduling Server element can be used to model tasks. The processor which is mapped to this task can be associated with the task using the Scheduler attribute of scheduling Server. This scheduler will use its scheduling strategies to assign the processing power associated with it to this scheduling server. The information required by the scheduler to make scheduling decisions for this task can be represented using the Server Scheduling Parameters attribute of the scheduling server. The type attribute of the server scheduling parameter should be set Fixed Priority Policy. Further, its Priority attribute can be used to specify the priority of the task. Since the priorities of tasks are statically defined in AUTOSAR OS, so to stop MAST from calculating the new priority of the associated scheduling server, the Preassigned attribute of the server scheduling parameter element must be set No. In the case of immediate priority ceiling protocol, the only information required from scheduling server is its priority. Consequently, the designer need not supply the Synch Parameters attribute of the scheduling server, which is used to supply additional synchronization information for the task to access the associated shared resources. A scheduling server gets the workload from an Operation element which is mapped to it using an Event Handler element. This is discussed in the next section.

3.3.3 Modeling the AUTOSAR Application

As mentioned before, an AUTOSAR application consists of SWCs and these SWCs implement their functionality using runnables. A runnable's implementation is very similar to a function. It has its own code and several calls to functions and RTE APIs, e.g., the code snippet of runnable swcHeadlightMainRunnable in the **Section 3.2** has its own code, a function call, and two RTE APIs calls. A runnable can be modeled as an Activity type Event Handler element of the MAST model. Its Operation attribute can have a reference of Composite type operation element. The Operation List attribute of this operation can contain the list of all the operations where each operation corresponds to either a piece of code of the runnable, a function call, or an RTE API call. The task that is mapped to this runnable can be specified in the Activity Server attribute of the activity element. Depending upon the AUTOSAR stack implementation, communication between the runnables could use shared buffers. In such cases, if the size of the buffer is greater than the word size of the ECU, each shared buffer should be modeled as a Shared Resource element as described in the last section. Moreover, a runnable is usually activated by an event and generates an event as its output. An event which starts a runnable can be modeled using an External Event element. Its attribute Type can be used to represent how this event should be generated, e.g. periodic, singular. An Internal Event element of the MAST model can be used to model an element generated by a runnable. External and internal events can be mapped to a runnable using the Input Event and Output Event attributes of the activity representing the runnable. It is important to note that a periodic runnable or any piece of code which gets executed after the event generated by the System Timer element should be modeled as a System Timed Activity type event handler. For a system timed activity, MAST implicitly considers the jitter caused by the system timer involved in activating the Event Handler.

Timing constraints on a runnable or any collection of code can be imposed using the events. First, the runnable or the piece of code has to be modeled as an activity as mentioned in the last paragraph. The next step involves modeling the internal event that the modeled activity will generate. The definition of the internal event will have a reference to an instance of Timing Requirement element that will contain the information of the imposed timing requirement. As a final step, the reference timing requirement element needs to be defined. The time limit of the timing constraint should be specified in the attribute Deadline. The strictness of the timing constraint can be specified using the "hard" or "soft" pre-defined values of the Type attributes of a timing requirement. If the deadline has to be imposed from the time the execution of the activity started till the associated event is generated, then Type attribute should be modeled as "local". Otherwise, if the deadline has to be imposed with reference to some other event then the Type attribute should be modeled as "global" and the referred event should be specified using the Referenced Event attribute.

Generally, runnables in an AUTOSAR application get executed one after another in a chain sequence. A system generated event, e.g., periodic timer expiry, activates a runnable and this runnable generates an event as an output that activates another runnable. This event chain can be represented using the Transaction element of MAST model. The External Events, Internal Events, and Event Handlers attributes of the Transaction element can be used to specify the runnable activating events, runnable generated events, and runnables, respectively.

Consider an example of AUTOSAR application. It has two SWCs. SWC1 has two runnables R1 and R2 while SWC2 has one runnable R3. R1 has to be activated periodically while R2 and R3 get activated by an event generated by R1 and R2, respectively. R1 and R2 are mapped to the same task T1 while R3 is mapped to another task T2. Additionally, R2 has a timing constraint that it should finish its execution in time TC1. The MAST model elements and their relations corresponding to the described AUTOSAR application is shown in the **Figure 3.6**. R1 is modeled as a Timed Activity type Event Handler Timed_Activity_1 that get activated by a Periodic type External Event Ex_Event_1. The events generated by R1 and R2 that activates R2 and R3 are modeled as Internal Events In_Event_1 and In_Event_1, respectively. The event generated by R3 is represented by In_Event_3. The timing restrictions of R2 are modeled using a Timing Requirement element Timing_Req_1 that is associated with the generation of the event In_Event_2. The Type of the Timing_Req_1 will be "local" and the attribute Deadline will have the value TC1. Furthermore, the runnable to task mapping is represented using the two Scheduling Server elements Sch_Server_1 and Sch_Server_2. R1 and R2 are mapped to Sch_Server_1 representing the task T1 while R3 is mapped to Sch_Server_2 representing the task T2. Finally, the two SWCs are represented using the two Transactions Transaction_1 and Transaction_2.

Table 3.1 represents the mappings of all the major element of an AUTOSAR system to their corresponding MAST element.

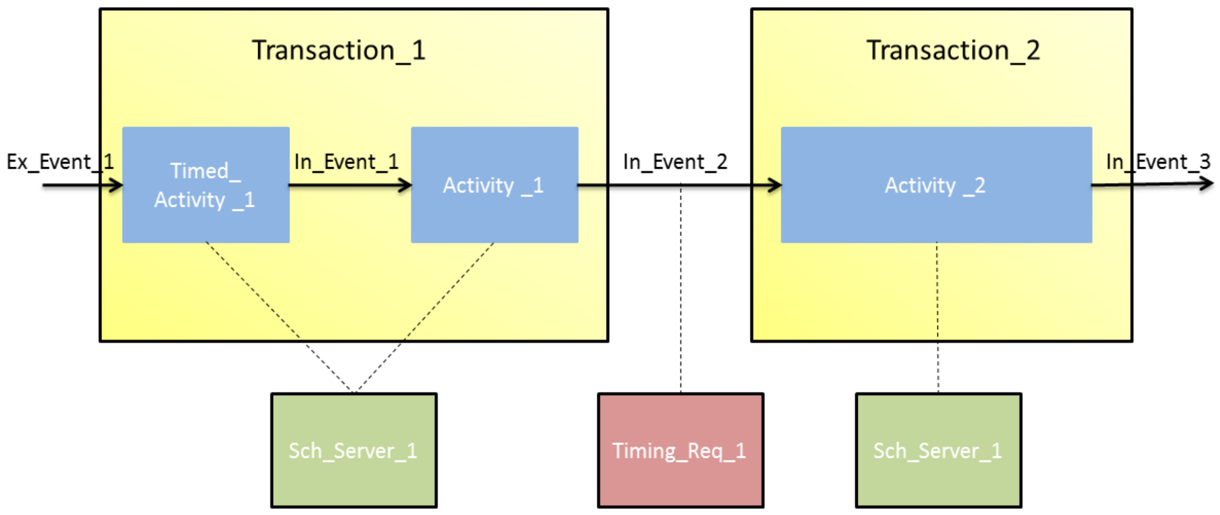


Figure 3.6: Example showing the modeling of runnables, tasks, and timing requirments.

AUTOSAR System Entity	**MAST Model Element**
CPU	Processing_Resource of type Regular_Processor
System timer	System_Timer of type Ticker
Network	Processing_Resource of type Packet_Based_Network
Operation done for handling incoming or outgoing packets	Netwrok Driver
Shared hardware resources	Shared_Resource
RTE APIs (with no critical section)	Simple or Composite Operations
RTE APIs (with critical section)	Composite Operations
Synchronization mechanism	Shared_Resource
Critical section	Operation with Shared_Resources of type Immediate_Ceiling_Resource
OS scheduler	Scheduler of type Primary_Scheduler
Scheduling algorithm	Scheduling_Policy of type Fixed_Priority
Tasks	Scheduling_Server with fixed priority
Runnable	Event_Handler of type Activity
Periodic runnable	Event_Handler of type System_Timed_Activity
Runnable activation events	External_Event
Events generated by a runnable	Internal_Event
Timing constraints of a runnable	Timing_Requirement
Event chain	Transaction

Table 3.1: Summary of AUTOSAR system entities to MAST model elements mapping

Using the mappings described in the above sections, the runnable swcHeadlightMainRunnable and its related AUTOSAR entities present in the code snippets in the **Section 3.2** are modeled into a MAST model. This MAST model is presented below. The time values used for various parameters are indicative.

```
Processing_Resource (
   Type                     => Regular_Processor,
   Name                     => cpu_1,
   Max_Interrupt_Priority   => 300,
   Min_Interrupt_Priority   => 250,
   Worst_ISR_Switch         => 186,
   Avg_ISR_Switch           => 186,
   Best_ISR_Switch          => 186,
  (Type => Ticker,
      Worst_Overhead => 3938,
      Avg_Overhead => 1191,
      Best_Overhead => 943,
      Period => 72000),
    Speed_Factor => 1.0);

Scheduler (
   Type              => Primary_Scheduler,
   Name              => scheduler_1,
   Host              => cpu_1,
   Policy            =>
      ( Type                    => Fixed_Priority,
        Worst_Context_Switch => 4559.0,
        Avg_Context_Switch   => 4514.0,
        Best_Context_Switch  => 4478.0,
        Max_Priority         => 300,
        Min_Priority         => 1));

Scheduling_Server (
   Type                          => Regular,
   Name                          => task_OsHTask,
   Server_Sched_Parameters       =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 8,
        Preassigned  => NO),
   Scheduler                     => scheduler_1);

Shared_Resource (
   Type               => immediate_Ceiling_Resource,
   Name               => buffer_lightBrightness,
   Ceiling       => 300,
   Preassigned        => YES);

Operation (
   Type                          => Simple,
   Name                          => OHTask_WaitEvent_to_Runnable,
    Worst_Case_Execution_Time  => 23.00,
    Avg_Case_Execution_Time    => 19.00,
    Best_Case_Execution_Time   => 18.00);

Operation (
    Type                       => Simple,
    Name                       =>
     Runnable_Begining_to_Communication_API_Critical_Section,
    Worst_Case_Execution_Time  => 112.00,
    Avg_Case_Execution_Time    => 109.00,
    Best_Case_Execution_Time   => 108.00);

Operation (
    Type                       => Simple,
    Name                       => Communication_API_Critical_Section,
    Worst_Case_Execution_Time  => 108.00,
    Avg_Case_Execution_Time    => 106.00,
    Best_Case_Execution_Time   => 104.00,
    Shared_Resources_To_Lock   =>
        ( buffer_lightBrightness),
    Shared_Resources_To_Unlock =>
        ( buffer_lightBrightness));

Operation (
    Type                       => Simple,
    Name                       =>
     Communication_API_Critical_Section_to_PWM_RTE_API,
    Worst_Case_Execution_Time  => 223.00,
    Avg_Case_Execution_Time    => 219.00,
    Best_Case_Execution_Time   => 218.00);

Operation (
    Type                       => Simple,
    Name                       => PWM_RTE_API,
    Worst_Case_Execution_Time  => 323.00,
    Avg_Case_Execution_Time    => 319.00,
    Best_Case_Execution_Time   => 318.00);

Operation (
    Type                       => Simple,
    Name                       => PWM_RTE_API_to_End_Of_Runnable,
    Worst_Case_Execution_Time  => 33.00,
    Avg_Case_Execution_Time    => 29.00,
    Best_Case_Execution_Time   => 27.00);

```

```
Operation (
   Type                        => Simple,
   Name                        => End_Of_Runnable_to_Wait_Function_of_OHTask,
   Worst_Case_Execution_Time   => 44.00,
   Avg_Case_Execution_Time     => 43.00,
   Best_Case_Execution_Time    => 43.00);

Operation (
   Type                      => Composite,
   Name                      => swcHeadlightMainRunnable,
   Composite_Operation_List  =>
      ( Runnable_Begineing_to_Communication_API,
        Communication_API,
        Communication_API_to_PWM_RTE_API,
    PWM_RTE_API,
    PWM_RTE_API_to_End_Of_Runnable
    ));

Operation (
   Type                      => Composite,
   Name                      => Executing_swcHeadlightMainRunnable,
   Composite_Operation_List  =>
      ( OHTask_WaitEvent_to_Runnable,
    swcHeadlightMainRunnable,
    End_Of_Runnable_to_Wait_Function_of_OHTask
    ));

Transaction (
   Type            => regular,
   Name            => transaction_headlight,
   External_Events =>
      ( ( Type       => Periodic,
          Name       => event_periodic_hl_50ms,
          Period     => 72000,
          Max_Jitter => 0.000,
          Phase      => 0.000)),
   Internal_Events =>
      ( ( Type => Regular,
          Name => internal_unused_event)),
   Event_Handlers  =>
      ( (Type                  => System_Timed_Activity,
         Input_Event           => event_periodic_hl_50ms,
         Output_Event          => internal_unused_event,
         Activity_Operation    => Executing_swcHeadlightMainRunnable,
         Activity_Server       => task_OsHTask)));
```

Chapter 4

Evaluation and Results

This chapter evaluates the method presented in **Chapter 3** using a use-case, and then explores the effects of different types of overheads using synthetic applications. To evaluate the proposed method, an AUTOSAR application Front Light Management was developed. The overheads for the AUTOSAR stack used in the experiment were measured, and then their effects on the Front Light Management were analyzed. Later, to generalize the effects of overheads on any application, synthetic applications were used. It gives an interesting insight on how different types of overheads affect an AUTOSAR application performance. The following sections describe this process in detail.

4.1 Research Questions

The intention behind performing this evaluation was to know the answer of following main questions:

1. Is it possible to perform schedulability analysis of an AUTOSAR application using the proposed method?
2. How much effect these overheads can cause on the schedulability of an AUTOSAR application?

4.2 Experimental Set-up

While performing the evaluation, many software tools and hardware devices were used. This section gives the details about all the software tools and hardware devices used, along with their arrangements.

4.2.1 ARCCORE

ARCCORE is an AUTOSAR products vendor and offers a complete suite required to develop an AUTOSAR application [1]. The two ARCCORE tools used in developing the AUTOSAR application are the following:

1. **Arctic Studio:** It is a tool chain, which provides the development environment and required tools for all the stages of the AUTOSAR based software applications [3]. The Arctic studio IDE is eclipse based. Main tools provided by Arctic Studio are [21],

   - SWC Builder - used to describe the components of the AUTOSAR based software application.
   - EXTRACT Builder – generates the application extract for the ECU.
   - RTE Builder – used to configure and generate the code for RTE layer.
   - BSW builder – used to configure and generate the code for BSW layer.

   The Arctic Studio version used in the experiment is ArcticStudio-arm-11.0.0 (64bit) which follows the AUTOSAR 4 specifications.

2. **Arctic Core:** It is ArcCore's AUTOSAR stack implementation [2]. It contains the GPL source code to support all the features required in an automotive ECU, e.g., communication services, diagnostic services, microcontroller specific code, operating system. The version used in experiment is core-v11_0_0 which is also AUTOSAR 4 based.

The process of developing the application starts by defining the descriptions of the software components of the automotive software application in the Arctic Studio. Software Component Language is the modelling language supported by the Arctic Studio, to model the SWCs. Software Component Language is one of the textual modelling languages for AUTOSAR, standardized by ARText Framework [4]. As discussed in **Chapter 2.1**, SWC descriptions contain runnable information, ports and interface used by an SWC, system services required by the SWC, implementation details, and other requirements. Based on the description of the SWCs, an application level composition is described using the same language. Once its validation is successful, extract for the ECU is generated. Depending on the requirements in this ECU extract, RTE, services required from BSW, and Operating System are configured. ArcCore provides inbuilt support for several ECUs, if the ECU being used is not supported, then one needs to provide the ECU resource description in a format specified by AUTOSAR. In the next step, all these configurations are validated and on successful validation, the code for all the required modules of AUTOSAR stack is generated. This generated code also contains the RTE level interface files. These files contain all the APIs which can be used during the implementation of the SWC, to access all the stack services, e.g., Rte_Call_HeadlightPwmDuty_Set(. . .), Rte_Read_SwitchStatus(..). Once the implementation of the SWCs is finished, it can be compiled with the rest of the generated code. After the successful compilation, the executable for the ECU will be generated. Now, this executable can be run on the targeted ECU in the vehicle to implement the required functionality.

4.2.2 Hardware and Other Tools

The hardware board used to run the automotive software application developed was STM32F 107VC. The key features of this ECU are [28]:

- Core: ARM® 32-bit Cortex® -M3 CPU, 72 MHz maximum frequency.
- Memories : 64 to 256 Kbytes of Flash memory, 64 Kbytes of general-purpose SRAM.
- DMA: 12-channel DMA controller.
- Supported peripherals: timers, ADCs, DAC, I2Ss, SPIs, I2Cs and USARTs.
- Serial wire debug (SWD) & JTAG interfaces.
- Up to 80 fast I/O ports.
- Up to 14 communication interfaces with pinout remap capability.

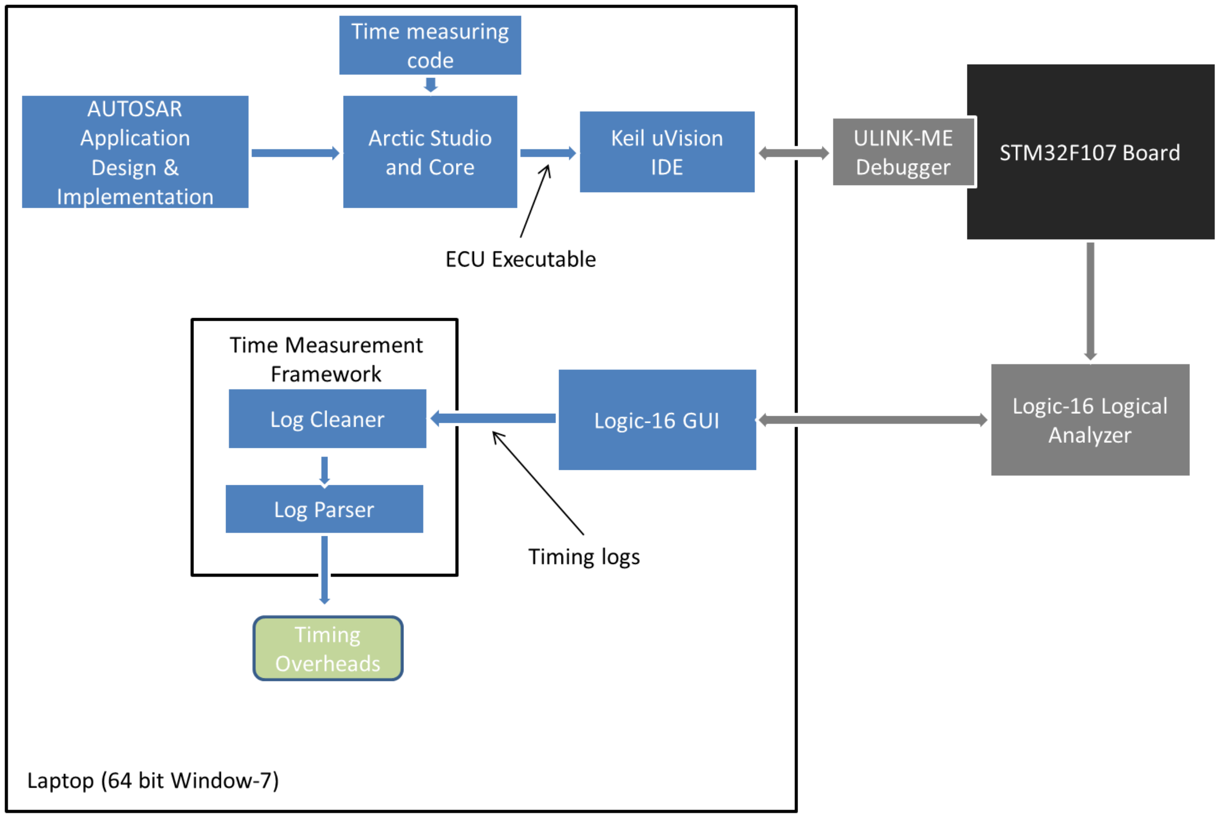


Figure 4.1: Experimental Set-up for timing measurement

To debug and download the executable, generated using ArcCore tool chain, Keil uVision5 (ARM Microcontroller Development Kit) was used [17]. It was connected to the STM development board using ULINK-ME debug and trace adaptor [16]. ULINK-ME is a JTAG debug interface based adapter.

To read the values from digital I/O pins of the STM board, Saleae's Logic16 Original logical analyzer was used [27]. It is a small size, GUI based, easy to use, logical analyzer device which can be used to analyze and record digital signal samples, at the maximum rate of 100 million samples per second. It can be connected to the pins available on the board using standard wires and to the computer, using provided USB cable. Once the device is connected, its GUI can be used from the computer, to configure all the available parameters, access and visualize logs, and perform other required operations.

All the tools were used from a laptop running 64 bit version of Windows 7. This entire set up of this experiment is shown in **Figure** 4.1.

4.2.3 A Method to measure the Code Execution Time

To measure the execution time of a function or a collection of code, the following method can be used. Any piece of code for which time needs to be measured has a start and an end. The start and end can be represented using an integer. When the duration starts, this value is written on DIO (Digital Input Output) pins, and when it ends, the same integer value is written again on the same pins. This integer value is limited by the maximum number which can be written on the DIO pins in the binary format. A code fragment, similar to the one shown below can be added in the AUTOSAR stack implementation to enable writing the start and the end of the time measurement on the DIO pins.

```
typedef enum MyEventType{
  EVENT_START = 0, //no value
  Event_1,
  Event_2,
  Event_3
}TimeEvent_E;

#define ADDRESS ((uint32_t*)(0x4001080C))
#define WRITE_EVENT_TIMESTAMP(event) (*ADDRESS) = event; \
                                      (*ADDRESS) = 0x0;
```

The start and the end of time measurement can be marked as follows,

```
WRITE_EVENT_TIMESTAMP(Event_1);
Rte_API_Call(...);
WRITE_EVENT_TIMESTAMP(Event_1);
```

A logical analyzer which connects to the same DIO pins where values are being written will store the timestamp along with the value whenever any value is written on these pins in a log file. Now, these log files first need to be cleaned to remove any erroneous values and then later simply parsed by checking the difference between the timestamp of the same integer value in pairs designating the start and end of a duration. This measured time also contains the time taken in writing the logged value on the DIO pins. For example, when the described method was used in the evaluation, the time taken in writing the logged value of DIO pins was 9 CPU cycles. This additional time taken needs to be subtracted from each of the measured time durations. Furthermore, the log parsing needs to be done over a number of files and ideally, should give three outputs – Best Case Time, Average Case Time, and Worst Case Time. This is the procedure used in the evaluation of this method and is described in **Section 4.3**. The framework which cleans and parses the time logs as described in this step of the method was implemented for the evaluation and can be accessed from this repository [13].

4.3 Measuring the System Overheads

The technique described in **Section 4.2.3** is used to measure the overheads for the executable which implemented the Front Light Management functionality with its entire AUTOSAR stack. The process started with finding all the places in the source code of the AUTOSAR stack where the timestamp needs to be logged. These are the places which mark either the start or the end of the execution of non-application code. A different integer value was logged for each of the overhead twice, designating the start and the end of the overhead. E.g.,

```
WRITE_EVENT_TIMESTAMP(Event_10);
OS_CALL_SuspendOSInterrupts();
WRITE_EVENT_TIMESTAMP(Event_10);

*value = Rte_Buffer_swcFrontLightManager_lightRequestReceiver;

WRITE_EVENT_TIMESTAMP(Event_11);
OS_CALL_ResumeOSInterrupts();
WRITE_EVENT_TIMESTAMP(Event_11);
```

These values were written to the four DIO pins which were connected to the logical analyzer. Logical analyzer recorded the timestamp for each of these values and stored them in a file. After collecting sufficient data, Time Measuring Framework was executed on these files. It first cleaned the recorded logs to remove erroneous data and then calculated the values of each of the overhead, by calculating the difference between two timestamps of the integer value in the logs, assigned for that particular overhead. All the sources code of timing measurement, framework developed to clean the logs and calculate timing overheads from these logs, can be found in the repository [13].

4.3.1 Overhead Measurement Results

Different type of overheads of AUTOSAR stack running on Arm Cortex-M3 core were measured for the use-case. These overheads are are listed in **Table 4.1**. The unit of time is CPU cycles for the entire experiment.

Overhead	Worst Case Time	Avg Case Time	Best Case Time
Context Switching	4559	4514	4478
System Timer	3938	1191	943
Interrupt Entry/Exit (ISR1)[1]	-	184/161	-
Interrupt Entry/Exit (ISR2)[1]	-	256/334	-
Pre-Runnable-Event	279	278	277
Post-Runnable-Event	40	40	38
Pre-Runnable (no implicit communication)	18	18	16
Post-Runnable(no implicit communication)	23	22	20
Synchronization C-S Server Call	158	157	153
DIO Write	1296	1262	1202
DIO Read	644	615	590
PWM Write	477	473	468
Explicit communication from runnable to critical section	36	35	32
Explicit communication from critical section to runnable	41	39	36
Explicit communication critical section (4 byte)	153	149	148
Explicit communication critical section (64 byte)	194	191	190
Explicit communication critical section (4096 byte)	3591	3591	3586

Table 4.1: Time(in CPU cycles) measured for different types of overheads.

The System Timer overhead is the sum of a tick interrupt entry and exit time, and the execution time of the timing ISR. The use-case has three periodic runnables and each of these runnables is mapped to a different task. Furthermore, use-case maintains three alarms, each of which generates an event on its expiry that in turn activates the mapped runnable's task to start the runnable execution. Additionally, none of the runnables use implicit communication. As mentioned in the **Section 3.1.2** In the case of implicit communication, all the messages dealt by a runnable are received and sent at the beginning and the end of the runnable, respectively. Due to this, the overhead caused by a runnable with implicit communication will depend on its degree of communication and hence will vary from case to case. For the runnables with no implicit communication, the runnable overhead and communication overhead are accounted separately. After analyzing the code

[1]Taken from the Arctic Core benchmark results received from its vendor.

generated for implicit and explicit communication, it is observed that the code generated for both is very similar. In the case of explicit communication, the pre and post runnable processing, and the communication code is placed at separate places while in the case of implicit communication, all the communication code becomes the part of the pre and post runnable processing. As a result, the sum of Communication, Pre, and Post Runnable overheads will remain same for both implicit and explicit communication. So, the total overhead of the system should be approximately the same, irrespective of implicit or explicit communication. Overhead in the case of synchronized client-server communication includes the time used in calling the server runnable and returning from the server runnable to the client runnable. DIO Read/Write and PWM overhead include all the overheads caused by the RTI API call used in the application to perform the intended operations. In this case, communication is implemented using the shared buffers, so the communication RTE API is divided into three parts and its overhead is listed for each part. **Figure 4.2** shows the explicit communication overhead for message size from 1 Byte to 4 KB. From the figure, it can be concluded that this overhead increases linearly with the increase in message size.

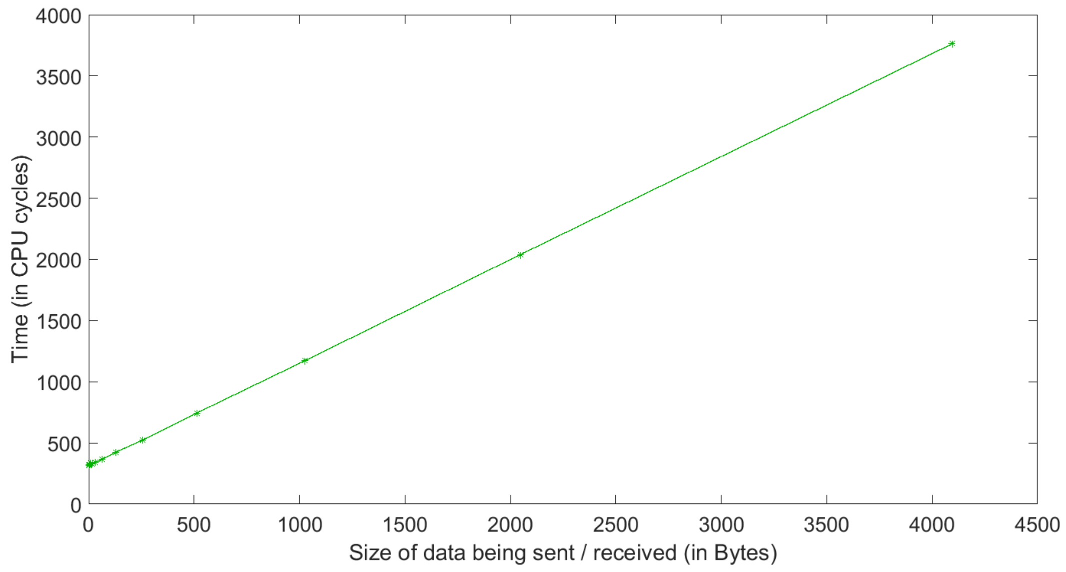


Figure 4.2: The graph representing the increase in the time of communication when message size increases. This matrix is for the explicit sender-receiver communication with no message queue and when the communicating runnables are mapped to the same task.

To avoid any spurious value, each of these overheads was measured for a total time of 240 seconds in 24 different occasions with each lasting for 10 seconds. For example,

Figure 4.3 shows the timer ISR execution times measured for the use case application used in the evaluation of the proposed method. For simplicity and clarity of the graph, only the first 1,000 values are drawn out of total 240,000 measured values. All the measured values are not same because the execution time of the timer ISR varies depending upon on the counters and alarms it has to maintain during its execution each time.

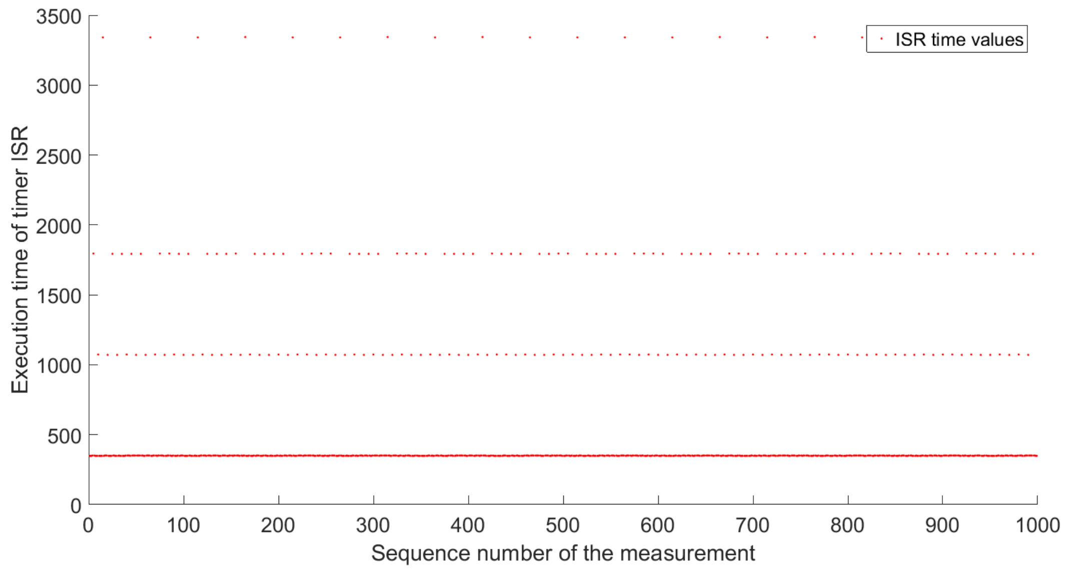


Figure 4.3: This graph shows only first 1,000 Measured overhead values for the timer ISR out of total 240,000 values.

4.4 Use-Case Evaluation

This is the realization of proposed method in **Chapter 3**. The following sections describe the details and the results of each of the steps of the proposed method, executed for the Front Light Management application.

4.4.1 AUTOSAR Application : Front Light Management

FLM (Front Light Management) is one of the most popular applications used in AUTOSAR specifications as an example. As the name suggests, this application implements the functionality to manage available lights in the front part of the vehicle. To replicate the behavior similar to actual Front Light Management application running in a vehicle, following resources from the STM board were used,

1. Two Buttons (marked as "key" and "temper" on the board). They will be addressed as B1 and B2.

2. Four LEDs (marked as LD 1, 2, 3 and 4 on the board). They will be addressed as LD1, LD2, LD3, and LD4.

The LEDs represented front lights of the vehicles and buttons acted as the available switches in the vehicle, which are used by the driver to control the lights. The LEDs mapping to the actual vehicle's lights and their respective states are shown in **Table 4.2**. Initial state for all the LEDs is "off". Button B1 is mapped to LD2 and can be used to change the state of LD2 to mimic the behavior of Headlight. Button B2 is connected to LD1, LD3, and LD4. It can be used to control the states of these LEDs to mimic the behavior of right indicator, left indicator, and daytime light.

LED Name	Replicated Light	States
LD2	Head Light	Off, Low Beam, High Beam
LD1	Right Indicator	Off, Blinking, On
LD3	Left Indicator	Off, Blinking, On
LD4	Daytime Light	Off, Blinking, On

Table 4.2: LEDs mapping to real vehicle lights and their states

The development of this application followed the steps described in **Section 2.1.4 and 4.2.1**. Initially, all application requirements were collected. Then the collected requirements were converted into the design. In the design, functionality of the Front Light Manager is implemented through four SWCs. **Figure 4.4** shows the design of these SWCs. The names of the runnables are shortened to accommodate them in the limited space. The details of each of these SWCs are following:

1. **SwcSwitchStatus:** The task of this SWC is to check if any of the buttons is pressed. This SWC has three runnables – initRunn, otherSwitchStatusRunn and headLightSwitchStatusRunn. initRunn initializes this SWC while headLightSwitchStatusRunn and otherSwitchStatusRunn observe if any event has occurred at B1 and

B2, respectively. headLightSwitchStatusRunn and otherSwitchStatusRunn are connected to B2 and B1 using the interfaces shown by <<1>> and <<2>>, respectively. These are the interfaces defined towards Digital Input Output(DIO) service provided by BSW through RTE generated APIs for this application. Using the client-server communication interfaces <<3>> and <<4>>, SWC SwcLightRequest invokes the client SWC SwcSwitchStatus to get the status of the two Buttons.

2. **SwcLightRequest:** This SWC has two runnables – initRunn and lightRequestRunn. initRunn initializes this SWC while lightRequestRunn manages the states of all the LEDs which can be changed by pressing the buttons. Once the state is updated as per the button event, SwcLightRequest sends the information of the mode and the brightness of each of the LEDs to the SWC SwcFrontLightManager using sender-receiver interfaces <<5>>, <<6>>, and <<7>>.

3. **SwcFrontLightManager:** This SWC has two runnables – initRunn and frontLightManagerRunn. initRunn initializes this SWC while frontLightManagerRunn turns the LD1, LD3, LD4 on and off, based on their respective brightness and mode. To control these LEDs, frontLightManagerRunn invokes the DIO server using its <<9>>, <<10>>, and <<11>> client-server interfaces, towards the BSW. For LD4, it does pre-processing of the data received from SWC SwcLightRequest, and sends it to the SWC SwcHeadlight through the <<8>> sender-receiver interface.

4. **SwcHeadlight:** This SWC has two runnables – initRunn and headLightRunn. initRunn initializes this SWC while headLightRunn calculates the PWM (pulse width modulation) duty cycles and controls the LD2 by invoking the PWM server using the client-server interface <<12>>. The PWM channel, for whom this duty cycle is being set, is configured in the BSW editor to connect it to the DIO channel of the LD2.

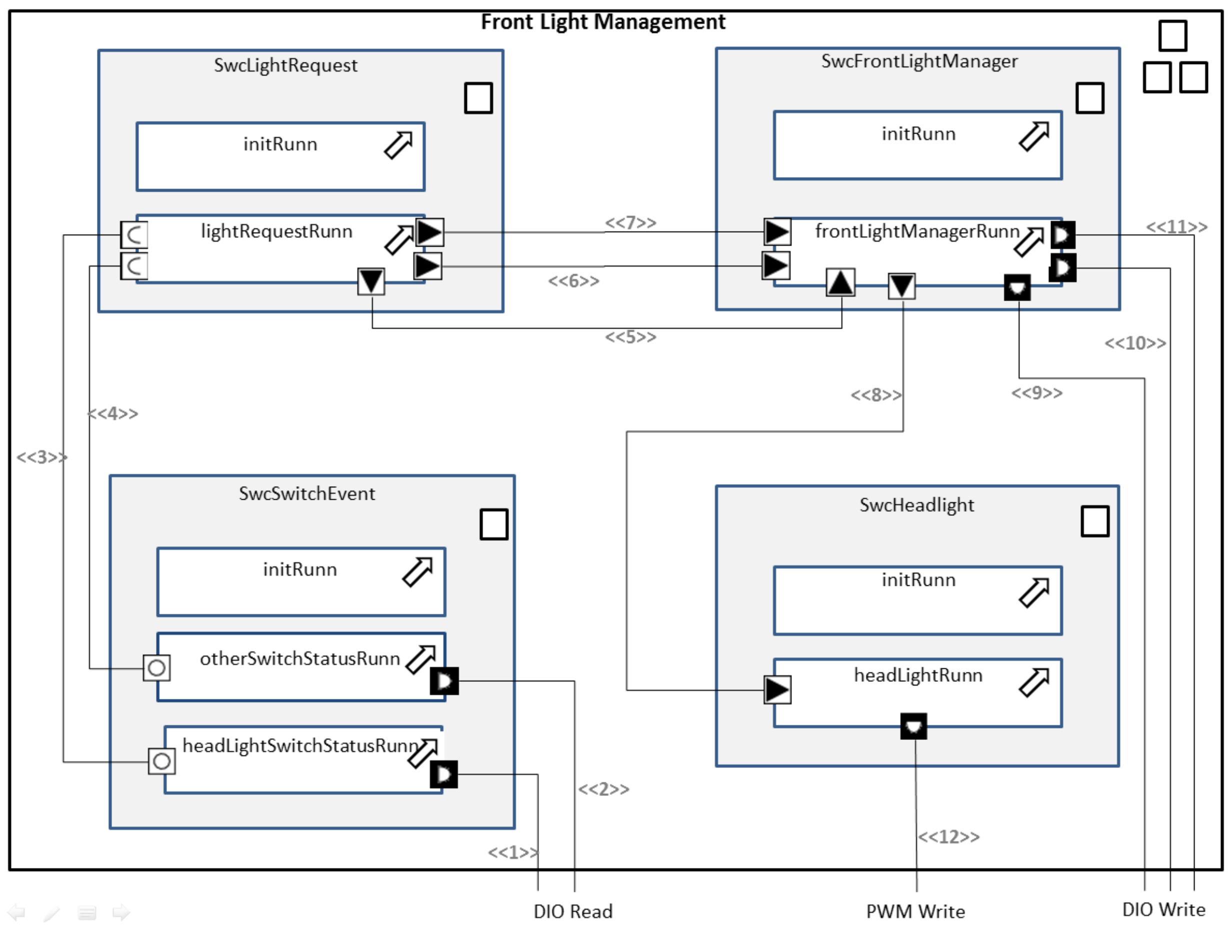


Figure 4.4: SWCs of the Front Light Management application. The numbers mentioned as <<digit>>represent the interfaces

Each of the the main runnables, lightRequestRunn, frontLightManagerRunn, and headLightRunn are mapped to three different tasks and are subscribed for the three different periodic events each having a period of 10 ms (72x10^4 CPU cycles). The priority of these three runnables's tasks is in following order lightRequestRunn > frontLightManagerRunn > headLightRunn. This discussed design of the the four SWCs was described using Software Description Language in Arctic Sdudio. **Appendix A** contains the complete description of all the four SWCs, their interfaces and ECU descriptions. As the next step, modules for all the required BSW services, RTE, and OS were configured using the BSW editor of Arctic Studio and the code for the AUTOSAR stack was generated. The RTE generated code contains all the APIs required by all the four SWCs, as per their specification mentioned in the description of their respective interfaces. These RTE APIs were used while implementing the SWCs to access system level services. In the final step, all the AUTOSAR stack generated code and SWCs implementation was compiled together to produce the executable for the ECU. The implementation of SWCs and complete configuration of BSW can be accessed from the cited repository [13].

4.4.2 Measuring the Front Light Management's Execution Time

As mentioned in the proposed method described in **Chapter 3**, one of the important parts of the schedulability analysis is the execution time of the SWCs of the application under analysis. To calculate the execution time of Front Light Manager, the method described in **Section 4.2.3** was used. This is the same method used in measuring the different types of overheads in **Section 4.3**. This execution time is purely the time taken by SWCs execution, i.e. all the RTE API calls for communication or services, and all the overheads should be excluded. Following is the code snippet of the implementation of the swcHeadlightMainRunnable runnable of SWC swcHeadlight. For the sake of simplicity some of the code is omitted.

```
void swcHeadlightMainRunnable(void){
 WRITE_EVENT_TIMESTAMP(Event_1); //Time_1_Start
  SignalQuality quality = 0;
  DutyCycle dutyCycle = 0;
  float64 headLightState = 0.0;
  Std_ReturnType ret = E_NOT_OK;
 WRITE_EVENT_TIMESTAMP(Event_1); //Time_1_End

  ret = Rte_Read_lightBrightnessReceiver_lightBrightness( &headLightState);

 WRITE_EVENT_TIMESTAMP(Event_2); //Time_2_Start
  if(RTE_E_OK != ret){
    ;//error handling code
  }
  if(BEAM_HIGH == headLightState)
    headLightState = VOLTAGE_MAX;
  else if(BEAM_LOW == headLightState)
    headLightState = (VOLTAGE_RANGE/2.0);
  else
    headLightState = VOLTAGE_MIN;
  ServoMotorControllerFunction(&headLightState, &dutyCycle); //local
    function call
  WRITE_EVENT_TIMESTAMP(Event_2); //Time_2_End

  Rte_Call_HeadlightPwmDuty_Set(dutyCycle, &quality);

  WRITE_EVENT_TIMESTAMP(Event_3); //Time_3_Start
  if(RTE_E_PwmServiceSetDuty_E_OK != ret){
    ;//error handling code
  }
  WRITE_EVENT_TIMESTAMP(Event_3); //Time_3_End
}
```

swcHeadlightMainRunnable is receiving one message and also calling RTE API to set the PWM duty of the headlight. These two API calls need to be excluded from the measurement. To achieve this, the time measurement for swcHeadlightMainRunnable is broken into three parts. These parts are marked as Time_1, Time_2, and Time_3 in the above code snippet. **Table** 4.3 shows the WCET (Worst Case Execution Time), ACET (Average Case Execution Time), and BCET (Best Case Execution Time) measured for each of these three parts.

Name of the parts	WCET	ACET	BCET
Time_1	3142	2840	2637
Time_2	86600	76391	68174
Time_3	1767	1763	1367
Total runnable execution time	91509	80994	72178

Table 4.3: Execution times (in CPU cycles) of different parts of swcHeadlightMainRunnable.

Similar to what described in the swcHeadlightMainRunnable example, the execution time for each of the runnables of all the four SWCs were measured. **Appendix** B contains the time measured for each part of all the runnables. A summary of this is presented in Table.

Name of the runnable	WCET	ACET	BCET
swcfrontlightmanager	548492	464900	230908
swclightrequestmainrunnable	35050	29053	26177
swcHeadlightMainRunnable	91509	80994	72178
swcheadlightswitchstatusrunnable	4344	3815	3290
swcotherlightsswitchstatusrunnable	4344	4015	3290

Table 4.4: Execution times (in CPU cycles) of the runnables

4.4.3 Executing MAST on MAST Model of Front Light Management

Using the method described in **Section** 3.3, a MAST model for the Front Light Management Application was created. This model described all the runnable operations, communications and other BSW services used by them. In the Front Light Manager's MAST model, each runnable is represented as a composite operation. Each SWC is also represented as a composite operation which is comprised of runnable's composite operations. The complete MAST model can be found in **Appendix B**. Each transaction in this model has a deadline equal to its period. For checking the schedulability and estimating the WCRT (Worst Case Response Time) of the application, this MAST model was executed using offset-based response time analysis technique. To identify the effects of overheads, this model was executed twice – with and without all the overheads. The results of this experiment are presented in **Tables 4.5** and **4.6**.

Model Configuration	**CPU Utilization**	**System Slack**	**Schedulable**
Without Overhead	93.76 %	6.25 %	Yes
With Overhead	103.03 %	-3.03 %	No

Table 4.5: Schedulability results of Front Light Manager with and without overheads

SWC Name	**Without Overhead**		**With Overhead**	
	Slack	WCRT	Slack	WCRT
SwcLightRequest	118.75 %	35164	-65.23 %	∞
SwcFrontLightManager	7.81 %	583651	-4.30 %	∞
SwcHeadlight	46.09 %	675052	-25.78 %	∞

Table 4.6: Slack and WCRT (in CPU cycles) for different SWCs

Initially, all the tasks of this application had a periodicity of 10 ms (72x10^4 CPU cycles) to make system more responsive to driver commands. After analyzing the results of the last run, it was observed that the application can become schedulable if the periodicity is increased to 50 ms (36x10^5CPU cycles). Since, FLM is not a very critical application, a periodicity of 50 ms will not cause any safety violation. Additionally, this increase in the system response time to driver commands is unnoticeable to the driver. Hence, the periodicity of all the tasks was changed to 50 ms in the FLM's MAST model. The results of executing the MAST on this changed model are listed in **Table 4.7** and **4.8**. As shown by the results, after making these changes the application became schedulable.

Model Configuration	**CPU Utilization**	**System Slack**	**Schedulable**
With Overhead (Second iteration)	24.98 %	300.22 %	Yes

Table 4.7: Schedulability results of Front Light Manager after increasing the task periods to 50 ms.

SWC Name	Slack	WCRT
SwcLightRequest	766.8 %	120219
SwcFrontLightManager	285.55 %	709328
SwcHeadlight	2898.8 %	817723

Table 4.8: Slack and WCRT (in CPU cycles) for different SWCs after increasing the task periods to 50 ms

4.4.4 Result and Discussion

The two executed scenarios (with and without the overhead), represent the cases when during the AUTOSAR application design, the overheads added by the AUTOSAR stack are considered or not. As shown in the **Table 4.5**, without overhead, CPU utilization is 93.76%, which gives enough room to either decrease the periods of the task of this application to make it more responsive to the user inputs or increase its features. Thinking this, an application designer might add more features to the application. But, the CPU utilization of the same Front Light Management application with the overhead, is 116.28%, which means this application is actually not schedulable. **Table 4.6** shows the slacks and WCRT (Worst Case Response Time) for individual SWCs. As reflected by the CPU utilization, when overheads are considered, the WCRT of the SWCs is infinite. This shows that this design configuration with these timing constraints for the Front Light

Management will not work, hence it needs to be changed. And the steps marked as 3 in the **Figure 3.1** need to be repeated, iteratively, until a schedulable design configuration is found.

Based on the inputs received from the first run, the application design configuration needed to be changed. Since the FLM's performance will still be acceptable if periods of its tasks are increased to 50 ms, so the periods of the tasks were increased to 50 ms in the application's MAST model. After all these changes, all the steps of the method which are marked as 3 were repeated. Since, with these change application became schedulable, this configuration was chosen as final design configuration for the FLM AUTOSAR application.

This evaluation of the proposed method answered all the questions raised in **Section 4.1**.

- (Q1) - Yes, it is possible to do schedulability analysis using the proposed method. During the evaluation, all the steps of the proposed method were successfully performed and in the end, a schedulable design configuration was found.
- (Q2) – The overheads are considerably large and can have measurable effects. As shown by the results of the evaluation, overheads can change the schedulability and performance of the overall system, drastically.

As shown in the evaluation of the method using FLM, apart from the schedulability information, there are some other important parameters in the result which can be used to get much more information about the system. E.g., system slack and CPU utilization can be used to see how much system load can be increased without affecting the system's shcedulability and performance, or how much it needs to be decreased to make the application schedulable. Slack and WCRT information about each SWC gives the information which of the SWCs are missing or meeting their deadline, and how much they need to be improved or can support more functionality, respectively.

4.5 Exploring the effects of Overheads using Synthetic AUTOSAR Applications

Section 4.4.4 described the effects of the overheads on a use-case application - Front Light Manager. The evaluation of Front Light Manager gives an idea about the impact of overheads. The goal of this section is to generalize the effects of the different types of overheads. The generalization of the overhead effects will require analyzing them over thousands of applications representing different design configuration. To establish the effects of different types of overheads on any AUTOSAR application, a framework to create MAST models for synthetic applications was developed. These synthetic applications were used to perform experiments to see how the schedulability of a particular design configuration changes, if some parameters from this configuration are varied. This gave an interesting insight into how and up to what extent, different types of system added overheads can affect the application performance. The output of this experiment can be used during the design phase of the AUTOSAR application. Following sections chronicle this in detail.

4.5.1 The GEP Framework

To generate synthetic AUTOSAR applications, a framework called GEP (Generator Executor Parser) was developed. It has several exposed APIs which can be used to tweak the parameters to of the design configuration of a synthetic application of a synthet, e.g., setnoOfTasks(). The GEP framework automated the task of generating the application MAST models, executing MAST over them, and then parsing the results, using following three components:

1. **Generator:** This is the component responsible for generating the MAST model for synthetic applications based on the parameters set. Due to the randomization involved in choosing various parameters of a design configuration and a large number of models generated for a set of parameters, the synthetic applications generated by this component covers a wide range of AUTOSAR applications.

2. **Executor:** This component executes the MAST tool over the generated MAST models and writes the results to an XML file in the configured directory. Various parameters for the MAST tool can be configured here.

3. **Parser:** This component parses the XML output of the MAST execution and produces the output as an object of data class which contains all the information about the models, e.g.- CPU Utilization, Worst Response Times. The main driving program uses this data object to give the percentage of schedulable models for a particular configuration.

Table 4.9 lists default values for of the configurable parameters of the GEP framework. One of the input parameters for the GEP framework is utilization factor. Application's utilization factor is the percentage of the total available computational power used by it. The utilization value is divided uniformly between the number of tasks configured using the UUniSort algorithm [12]. Next, the period of each task is selected randomly within the range configured for the periods of the tasks. For each task, its period is multiplied by its utilization factor to get the execution time for this task. For all the tasks, their period is also their completion deadline. The priority of the tasks is assigned rate monotonically [19].

In the next step, the number of runnables mapped to one task is selected randomly within the configured range. Additionally, for each of these runnable, their message out degree is selected randomly from the configured range. The receivers of these messages are the runnables which are mapped to tasks other than the sender runnable's task. The receiver runnables are also selected randomly. Furthermore, for each of these messages, its size is selected randomly within the configured range for the message size. The number of IO hardware operations for a runnable are selected randomly within the configured range of the number of IO hardware operation per runnable. After fixing all these parameters for a task, the overhead due to each of them is added to the task's execution time. This completes all the information required for a synthetic AUTOSAR application. In the final step, the synthetic AUTOSAR application is converted into a MAST model by following the methodology described in **Section 3.3**. Later, MAST is executed on this model and then its results are parsed and presented in an easy to understand format. All the source code of GEP framework can be found in the repository [13].

Parameter	**Value or Range**
Number of tasks	8
Periods of the tasks	[1, 1x10^3]ms or [72x10^3, 72x10^6]CPU cycles
Number of runnables per task	[1, 16]
Number of IO Hw Operations (DIO/AIO/PWM) per runnable	[0,1]
Message out degree for a runnable	[0, 2]
Size of a messgae	[1B,1KB]

Table 4.9: Default configuration used to generate synthetic AUTOSAR applications

4.5.2 Results and Discussion

The objective of this experiment was to model the effect of each type of overhead individually. To achieve it, a general configuration for the synthetic AUTOSAR application was fixed; and only one parameter causing overhead was varied. The general configuration used in the experiment is the default configuration which is listed in the **Table 4.9**. As mentioned in the last section, every configuration of the synthetic application involved some randomization, e.g., periods of tasks and number of messages being sent by a runnable to the runnables mapped to different tasks. This randomization helped in covering the wide range of applications through the synthetic applications. Each configuration was run with utilization factor from 0.04 to 1 with a step of 0.05. Furthermore, for each utilization factor of each configuration, 2500 synthetic application models were generated and analyzed. For each executed case, the experiment was repeated twice. In the first run, all types of overheads were considered and in the second run, all types of overheads were ignored. The output of each experiment is shown using their respective graphs. In these graphs, the y-axis represents the percentage of synthetic applications which are schedulable out of the total synthetic applications generated for a configuration; the x-axis represents the utilization factor of a synthetic application. Following sections model the effects of different types of overheads when one of the parameters mentioned in **Table 4.9** is varied.

4.5.2.1 Varying the Number of Tasks in a System

In general, application designers have flexibility in mapping the runnables to the tasks. Due to this flexibility, the number of tasks in an application can be vary depending upon the choices made by the application designer. In trying different design configurations, the application functionality remains the same; this means that the utilization factor of the application will remain the same. While depending on the number of tasks in the application, the system overhead can vary. This experiment was performed to see the effects of system overhead on an application when the number of tasks in its design configuration is varied. **Figure 4.5** shows the results of this experiment. As shown in the figure, the percentage of schedulable synthetic applications decrease when the number of tasks is increased from 4 to 16, and then later to 64. This effect is same for both the cases when the experiment was conducted with and without the system overhead. However, one important difference between these two cases is that for the same design configuration and utilization factor, the percentage of schedulable synthetic applications is significantly higher in the case when the system overhead was not considered. For the no-overhead case, the increase in the number of tasks decreased the percentage of schedulable applications because when the tasks are scheduled rate monotonically, schedulability decreases with the increase in the number of tasks [19]. For the overhead case, the additional reason for the decrease in the percentage of the schedulable application with the increase in the number of tasks in the application is the AUTOSAR stack added overhead. The overhead comes mainly from the context switching and other task management required running and maintaining the tasks in the system, e.g., if there are more periodic tasks, then the number of timers required to activate them will be more. Additionally, since the number of runnables is defined per tasks, the increase in the number of tasks will increase the number of runnables in the application. This increase in the number of runnables will cause the increase in the number of messages and IO operations because these parameters are defined per runnable. As a result, with the increase in the number of tasks in an application, the runnable, communication and IO hardware operation overhead will also increase. This added overhead also caused the drop in the percentage of schedulable applications when the number of tasks is increased.

The lesson learned from this experiment is that the design configuration which requires lesser context switches should be preferred over another. One way to achieve this is by mapping the runnables with the same period to one task.

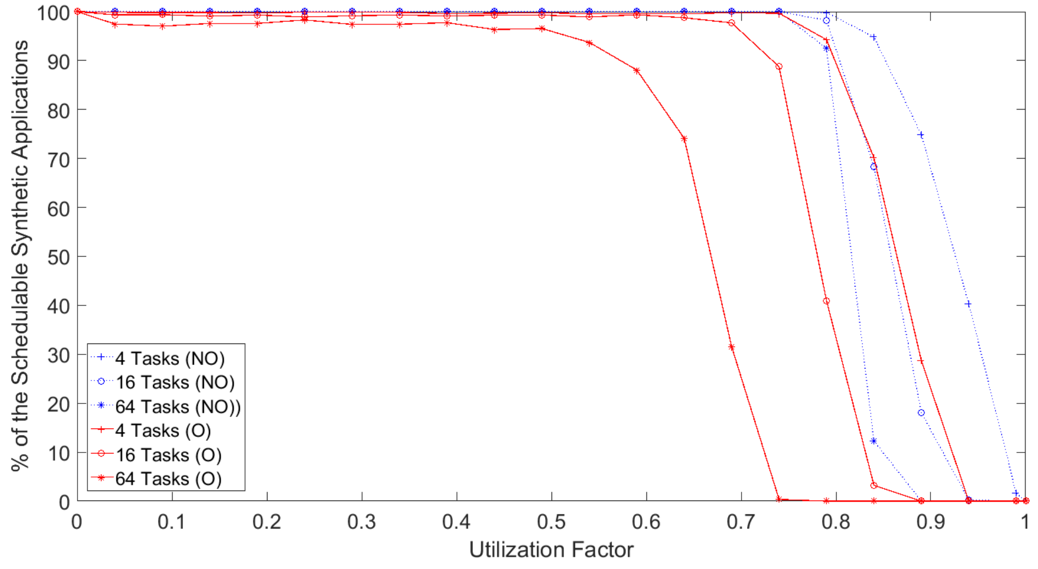


Figure 4.5: Effect of the overhead when number of tasks in an application are varied. “O” and “NO” represent the cases executed with and without overhead, respectively.

4.5.2.2 Varying number of Runnables mapped to a Task

This experiment was executed to model the effects of runnable overhead on an AUTOSAR application. In this experiment, the number of runnables mapped to a task was not randomly generated but fixed. Initially, the RPT (runnables per task) were fixed to 4 and then were increased to 16 and 64. The results of this experiment are presented in

Figure 4.6. While increasing the number of runnables, the rest of the configuration was kept same so that the total workload due to the factors other than number of runnables in the application remains same. For example, even if the RPT are changed from 4 to 16 or 64, the total number of messages being sent in the entire synthetic application will still remain same. From this graph, it can be concluded that runnable overhead alone does not affect the application's schedulability for at least up to 64 RPT. The reason behind this result can be explained by the value of the runnable overhead presented in **Table 4.1** which is very small to cause a significant effect.

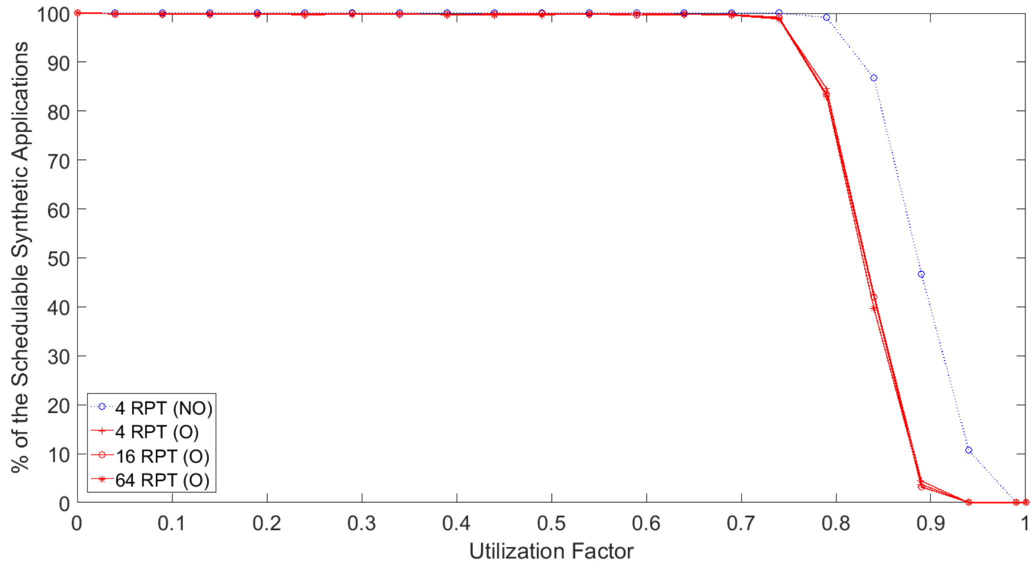


Figure 4.6: Effect of the overhead when the RPT (runnables per task) are varied. "O" and "NO" represent the cases executed with and without overhead, respectively.

4.5.2.3 Varying the range of the Periods of the periodic Tasks

This experiment was executed to check the effects of system overhead when the range of the periods of the periodic tasks in the system is varied. The ranges of periods considered are 1ms to 10ms, 10ms to 100ms and 100ms to 1000ms. The results of this experiment are presented in **Figure 4.7**. The percentage of schedulable applications for the same configuration is decidedly more for the period ranges [100ms, 1000ms] than the [10ms, 100ms]. However, the percentage of schedulable applications for the period range [1ms, 10ms] is drastically low. There are three main reasons behind this. First, with shorter periods, there will be more context switching and timer expiry overheads. Second, all the overhead added by AUTOSAR stack, e.g., communication, IO operations, and OS overhead, becomes relatively large when the period is short. The third and final important factor that affected the percentage of schedulable applications for the period range [1ms, 10ms] is the 1 ms tick period of the system timer. Any deviation from a normal expected value is called jitter. The jitter because of the tick period is at least 10% to maximum 100% for this period range. This jitter is very large and thus is capable of reducing the schedulability of the synthetic applications greatly.

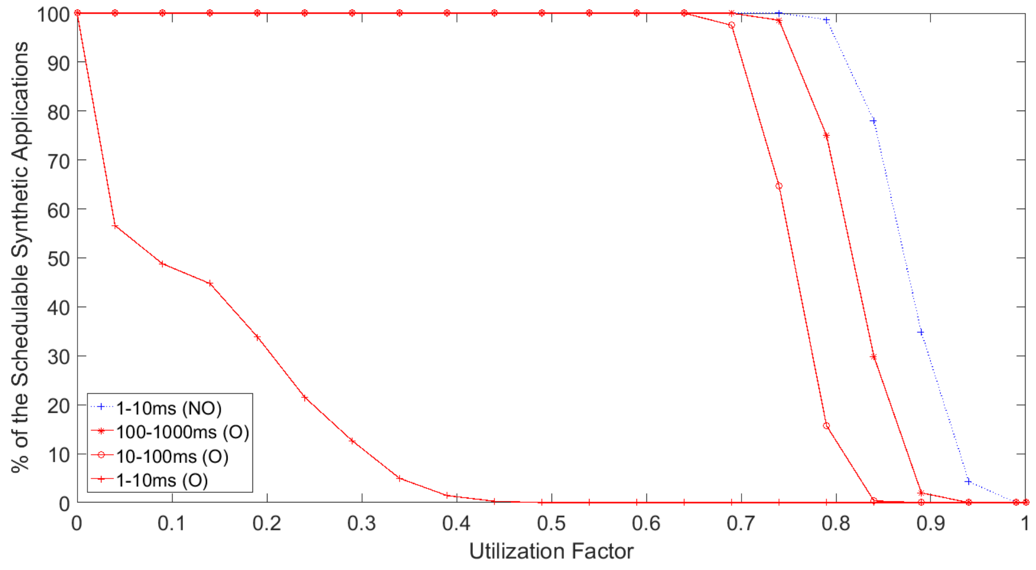


Figure 4.7: Effect of the overhead when range of the periods of the periodic tasks in an application is varied. The ranges [1,10] [10,100] and [100,1000] when changed from ms to CPU cycles are [72x10^3,72x10^4], [72x10^4,72x10^5] and [72x10^5,72x10^6]. “O” and “NO” represent the cases executed with and without overhead, respectively.

4.5.2.4 Varying number of IO Hardware Operations per Runnable

This experiment was conducted to model the effect of hardware IO overheads. The hardware IO overheads considered are analog IO, digital IO and PWM IO. In the overhead classification presented in **Section 3.2**, these overheads fall under the RTE API overheads. In this experiment, the IOPR (IO operations per runnable) is not randomly generated between the default range but fixed. For a fixed number of IO hardware operation, its type (analog IO, digital IO and PWM IO) was randomly selected. Initially, IOPR were fixed to 1 and then later were increased to 4 and 16. The results of this experiment are presented in **Figure 4.8**. Since for the no-overhead cases, the plots for the cases were identical, only one line is drawn to represent all of them. When the system overhead was considered, the percentage of schedulable applications decreased gradually from the 1 IOPR to 4 IOPR and then later to 16 IOPR. The reason behind this is the IO overhead values. These overheads values are large enough to cause deciding effects when the IOPR is increased.

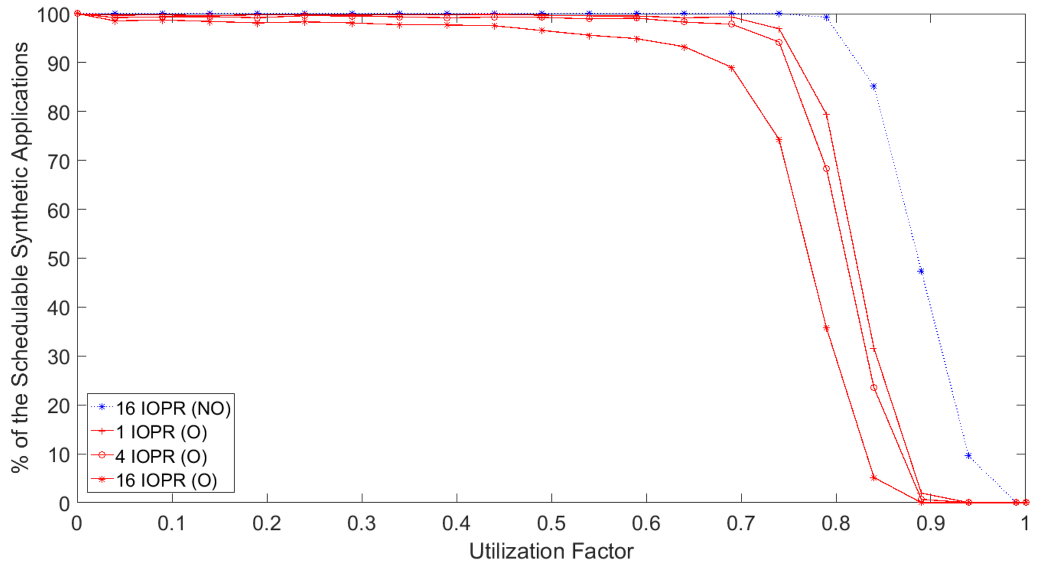


Figure 4.8: Effect of the overhead when the IOPR (IO operations per runnable) are varied. "O" and "NO" represent the cases executed with and without overhead, respectively.

4.5.2.5 Varying the Message Out Degree of a Runnable

This experiment was conducted to model the effects of communication overhead on an application's schedulability when the degree of communication is varied. In this experiment, the message out-degree of a runnable was fixed and its receiver was selected randomly among all the runnables which are mapped to tasks other than the sender's task. Initially, MPR (messages per runnable) were fixed to 1 and then later were increased to 4 and 16. The results of this experiment are presented in **Figure 4.9**. Since, for no overhead case the plots for 1, 4 and 16 MPR were identical, only one line is drawn to represent both of them. When the system overhead was considered, the percentage of schedulable applications decreased gradually from 1 MPR to 4 MPR and then later from 4 MPR to 16 MPR. The communication in the synthetic application is implemented using shared buffers. There are two reasons which caused the increased overhead when MPR are increased. First, copying more buffers takes more time. Second, in the MAST model of the synthetic applications, each shared buffer bigger than 32B is modeled as a different critical section with its unique Operation and Shared Resource element. From the graph, it can be concluded that increasing message out degree of a runnable in an AUTOSAR application can have significant effects on its schedulability.

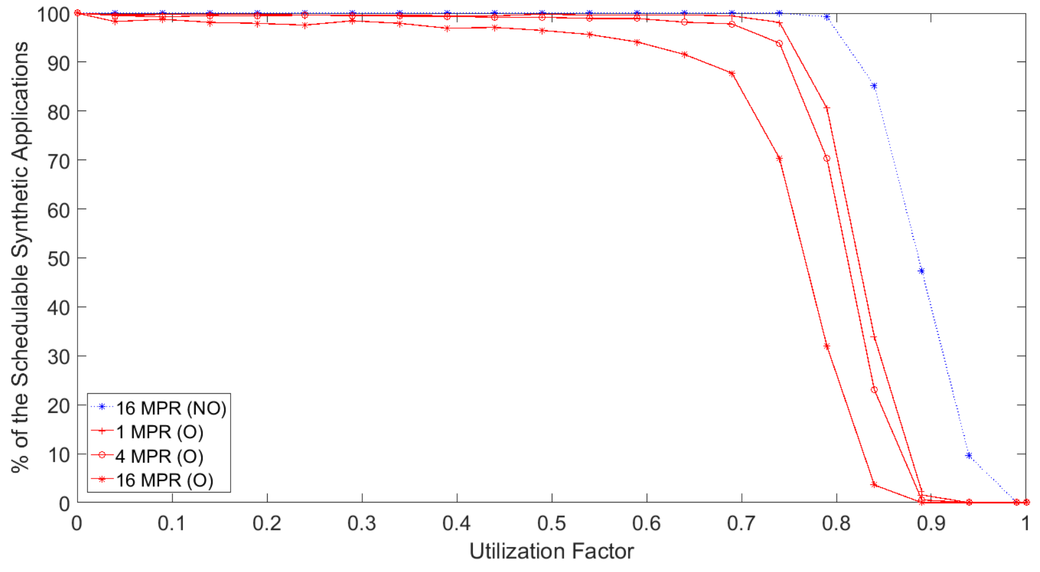


Figure 4.9: Effect of the overhead when the MPR (messages per runnable) are varied. “O” and “NO” represent the cases executed with and without overhead, respectively.

4.5.2.6 Varying the size of the data of a Message

This experiment was conducted to model the effects of communication overhead on an application’s schedulability when the message size is varied. In this experiment, the message size was fixed. Initially, it was set to 1 byte and then later, it was increased to 4 KB. The results of this experiment are presented in **Figure 4.10**. When the system overhead was not considered, the plot remained same for all considered values of message out degree. So, only one line is drawn to represent the no-overhead case. With the system overhead, the percentage of schedulable applications did not change significantly when the message size was increased from 1B to 4KB. From this graph, it can be concluded that the size of the message alone does not affect the application schedulability significantly. The reason behind this result can be explained by the small amount of time taken in copying the memory buffers of the larger sizes.

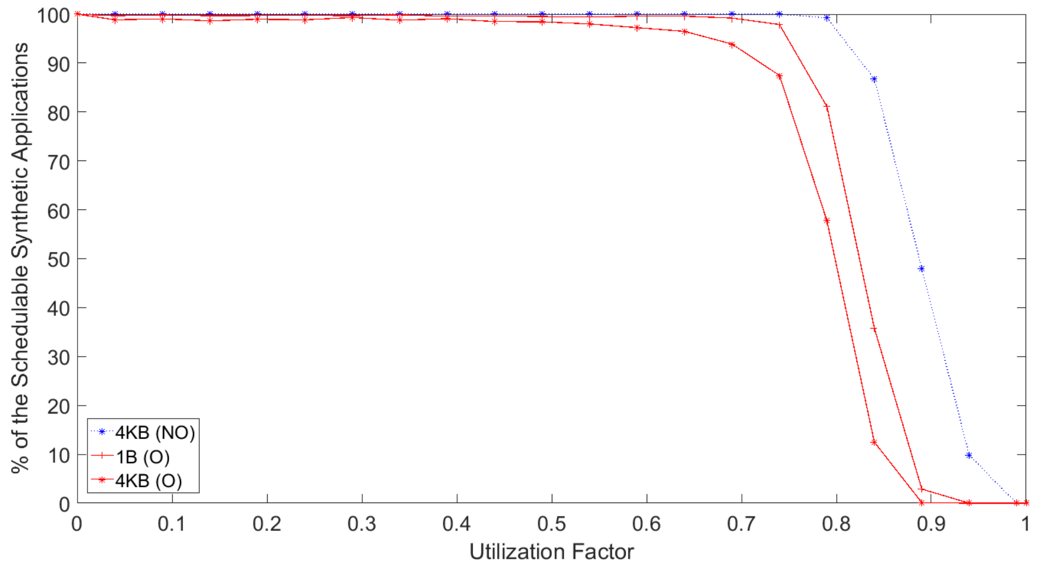


Figure 4.10: Effect of the overhead when the message sizes are varied. “O” and “NO” represent the cases executed with and without overhead, respectively.

4.5.2.7 Overhead if messages are being sent individually or using a Structure

This experiment was performed to check the communication overhead if a message is being sent as a structure and when all its elements are sent individually. In this experiment three cases were considered. The first case is when one message was sent for each byte of the data which need to be sent. In the second case, the message data was divided in random size chucks and then one message was sent for each of these data chunks. In the third case, entire message data was packed in one structure and then it was sent in one message. **Figure 4.11** shows the results of this experiment. As it can be seen in the figure, for the first case the communication overhead increases exponentially with the increase in the size of the data. In the second case, the overhead is still very large when compared to the results of the third case. In the third cases, the communication overhead increases linearly with the increase in the size of the message data. From this result, it can be concluded that among the considered cases, communication overhead is minimum when all message data is packed into a structure and sent in one message.

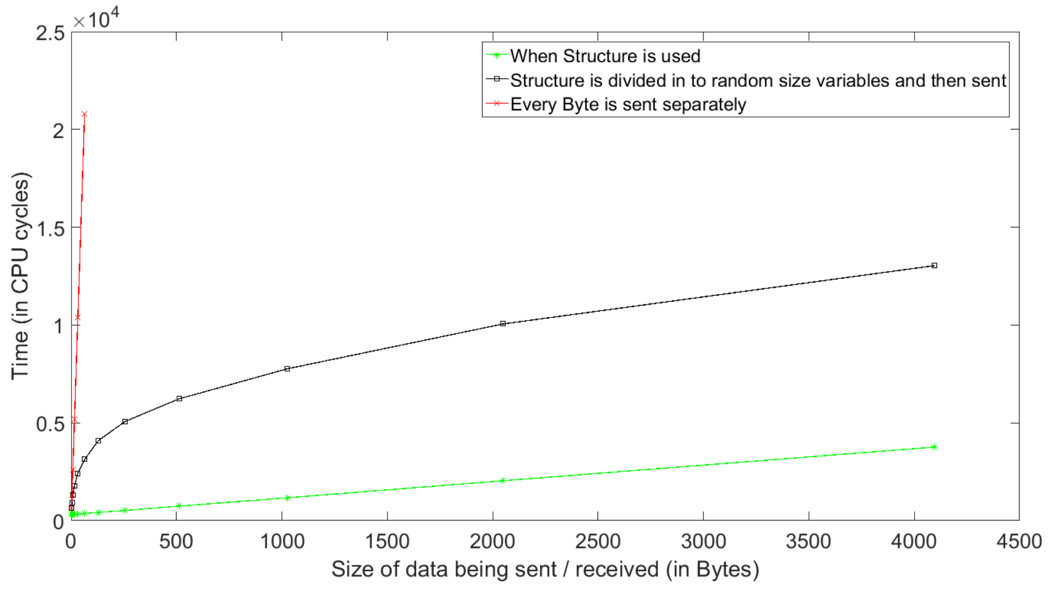


Figure 4.11: Effects of how data is being sent on communication overhead

This experiment also answers the question 2 raised in the **Section** 4.1. In all the executed cases, it was observed that irrespective of the parameter being varied, for the same design configuration and utilization factor, the number of schedulable applications is significantly lower when the system overhead is considered than the case when the experiment was performed without considering system overhead. **Table** 4.10 has listed the summary of this experiment.

4.6 Threats to Validity

The evaluation of the proposed method was done using a concrete use case. Later, the effects of different types of overheads were generalized using synthetic applications. In all these experiments, the ECU and the AUTOSAR stack used were always the same. Different ECUs have different architecture, processing power, and hence can give different results even if everything else is kept same. Similarly, many vendors supply different AUTOSAR stack generating tools. Even though they follow same AUTOSAR standard,

Parameter Varied	**Effects**
Number of tasks	Significant
Number of runnables per task	Almost no effect up to 64 RPT
Range of periods of tasks	Drastic for range [1 ms,10 ms] Significant for range [10 ms,1000 ms]
Number of IO hardware operations	Significant
Message out degree of runnable	Significant
Message Size	Moderate
Messages being sent individually or in a structure	Significant

Table 4.10: Summary of the effects of system overhead when a parameter from a fixed design configuration of an AUTOSAR application is varied.

their implementations still might be different, and hence can give different performance. So, the results presented might not be same for different ECUs and AUTOSAR stack implementations. During the experiment, applications did not use communication over networks. Also, the SWCs of the application were executed on the same core. So, the overheads and hence results might be different, if the application is hosted on multiple ECUs and communicate over the network.

Chapter 5

Conclusion and Future Work

In this thesis, an overhead-aware method to find the schedulable design configuration of a given AUTOSAR application is presented. Among the various steps of the proposed method, the main focus was on identifying and measuring the different types of system added overheads, and representing an AUTOSAR system as a MAST model. Later, the proposed method was evaluated using a sample AUTOSAR application to verify its applicability. This evaluation using a use case proved that a schedulable design configuration of an AUTOSAR application can be found by following the mentioned steps of the proposed method. During this evaluation, the overheads of the AUTOSAR stack were quantified, which gave an estimate of the values of different types of overheads. This evaluation also proved that different types of overheads added by the AUTOSAR stack can have significant effects on the schedulability of the application. To generalize the effects of various types of overheads, one more evaluation was performed. During this evaluation, to cover different configurations and types of AUTOSAR application, synthetic AUTOSAR applications were generated and analyzed using GEP framework. The effect of each type of overhead is modeled and presented in various graphs. The results of this experiment also iterated that the system added overheads can decidedly change the schedulability of an AUTOSAR application. This suggests that if the system added overhead is considered in the early stage of the AUTOSAR application development, then it will save the efforts, time, and cost which can otherwise be wasted if the application is found not schedulable in the later stages of its development. Moreover, the method can also be used to select the most suitable AUTOSAR stack implementation and hardware for the application.

In the future, the details about other steps of the proposed method will be presented. Given the AUTOSAR stack overhead, application timings, and hardware capabilities, a framework can be developed to automate all the steps to give the best design configuration for the application which is schedulable and meets all the mentioned constraints. This framework will be able to convert an AUTOSAR application into a MAST model and then performing the design exploration for it. Also, the generalization of the different types of overheads can be extended to include all network-related overheads. In this experiment, the applications considered were running on a single core. So the work can be extended to an application hosted on multiple cores on the same and different ECUs. Furthermore, in the overhead modeling using the synthetic applications, only a hardware device and an AUTOSAR stack was considered; so the effects can be generalized even more by performing these experiments with different combinations of the AUTOSAR stack implementation and the ECUs.

References

[1] ArcCore. http://www.arccore.com/. Accessed: 2016-07-30.

[2] ArcCore. Arctic Core. http://www.arccore.com/products/arctic-core. Accessed: 2016-07-30.

[3] ArcCore. Arctic Studio. http://www.arccore.com/products/arctic-studio. Accessed: 2016-07-30.

[4] Artop. Software component language. https://www.artop.org/artext/. Accessed: 2016-07-30.

[5] AUTOSAR. http://www.autosar.org. Accessed: 2016-07-30.

[6] AUTOSAR. AUTOSAR Timing Analysis Specification. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/methodology/auxiliary/AUTOSAR_TR_TimingAnalysis.pdf. Accessed: 2016-07-30.

[7] AUTOSAR. Basic Software Specification. https://www.autosar.org/fileadmin/files/releases/4-0/methodology-templates/templates/standard/AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf. Accessed: 2016-07-30.

[8] AUTOSAR. OS specification. https://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/auxiliary/AUTOSAR_SRS_OS.pdf. Accessed: 2016-07-30.

[9] AUTOSAR. Runtime Environment Specification. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf. Accessed: 2016-07-30.

[10] AUTOSAR. Software Component Template. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR_TPS_SoftwareComponentTemplate.pdf. Accessed: 2016-07-30.

[11] AUTOSAR. Virtual Function Bus Specification. http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR_EXP_VFB.pdf. Accessed: 2016-07-30.

[12] E. Bini and G.C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.

[13] Manish Chauhan. Source code for the autosar application front light manager, timing measurement framework, and gep syntactic mast model framework. https://bitbucket.org/iammanish/thesis, 2016.

[14] L. Marce F. Singhoff, J. Legrand, L. Nana. Cheddar : a flexible real time scheduling framework. *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time and distributed systems using Ada and related technologies*, pages 1–8, 2004.

[15] Object Management Group. Uml profile for marte^{TM}: Modeling and analysis of real-time embedded systems^{TM}. 20011.

[16] Keil. Ulink-me debugger. http://www2.keil.com/mdk5/ulink. Accessed: 2016-07-30.

[17] Keil. uVision IDE. http://www.keil.com/download/product/. Accessed: 2016-07-30.

[18] L. Sha, R. Rajkumar, J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[19] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machiner(JACM)*, 20(1):46–61, May 1973.

[20] MAST. http://mast.unican.es/. Accessed: 2016-07-30.

[21] Jesper Melin and Daniel Boström. Applying autosar in practice. http://www.idt.mdh.se/examensarbete/index.php?choice=show&id=1171, 2011.

[22] Abdollah Safaei Moghaddam. Performance evaluation and modeling of a multicore autosar system on theoretical modelling of speedup gain in heterogeneous multicore systems. 2014.

[23] M. Gonzalez Harbour, J.J. GutiCrrez Garcia, J.C. Palencia GutiCrrez, J.M. Drake Moyano. MAST: modeling and analysis suite for real time applications. *ECRTS '01 Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 125, 2001.

[24] Mehdi Aichouch, Jean-Christophe Pr´evotet, Fabienne Nouvel. Evaluation of the overheads and latencies of a virtualized rtos. *8th IEEE International Symposium on Industrial Embedded Systems*, 2013.

[25] OSEK. Operating System. http://www.osek-vdx.org. Accessed: 2016-07-30.

[26] Sorin Manolache, Petru Eles, Zebo Peng. Schedulability analysis of applications with stochastic task execution times. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2), 2004.

[27] Saleae. Logic-16 original logical analyzer. https://www.saleae.com/originallogic16. Accessed: 2016-07-30.

[28] STM. Stm32f-107vc development board. http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f1-series/stm32f105-107/stm32f107vc.html. Accessed: 2016-07-30.

[29] Saoussen Anssi, Sara Tucci-Piergiovanni, Stefan Kuntz, Sébastien Gérard, François Terrierl. Enabling scheduling analysis for autosar systems. *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), year =*.

[30] TIMMO. Timing Model - TOols, algorithms, languages, methodology, USE cases. https://itea3.org/project/timmo-2-use.html, 2013. Accessed: 2016-07-30.

[31] Ken Tindell. Adding time-offsets to schedulability analysis. *Technical Report YCS 221*, 1994.

[32] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Parallel processing in embedded real-time systems*, 40(2), 1994.

[33] P. E. Hladik, A. M. Deplanche, S. Faucou, Y. Trinquet. Adequacy between autosar os specification and real time scheduling theory. *International Symposium on Industrial Embedded Systems*, 2007.

APPENDICES

Appendix A

SWC Descriptions of Front Light Management Application

A.1 SWC SwcSwitchEvent

```
package SwcSwitchEvent

import Interfaces.*
import ArcCore.Services.IoHwAb.*
import Prop_Generic.Services.IoHwAb.*
import AUTOSAR.Services.EcuM.EcuM_CurrentMode
import AUTOSAR.Services.EcuM.EcuMFixedTypeMappings

component application SwcSwitchEventType{
  ports{
    client OtherLightsSwitchStatusClient requires DigitalServiceRead
    client HeadLightSwitchStatusClient requires DigitalServiceRead
    server OtherLightsSwitchStatusServer provides
   SwitchEventLightRequestCSIf1
    server HeadLightSwitchStatusServer provides SwitchEventLightRequestCSIf2
    receiver Mode requires EcuM_CurrentMode
  }
}

internalBehavior SwcSwitchEventBehavior for SwcSwitchEventType {
  dataTypeMappings {
    EcuMFixedTypeMappings
  }

  runnable SwcOtherLightsSwitchStatusRunnable [0.0] {
    symbol "swcOtherLightsSwitchStatusRunnable"
    server CallPoint synchronous OtherLightsSwitchStatusClient.Read
    operationInvokedEvent OtherLightsSwitchStatusServer.
   readOtherLightsSwitchStatus
  }

  runnable SwcHeadLightSwitchStatusRunnable [0.0] {
    symbol "swcHeadLightSwitchStatusRunnable"
    server CallPoint synchronous HeadLightSwitchStatusClient.Read
    operationInvokedEvent HeadLightSwitchStatusServer.
   readHeadLightSwitchStatus
  }
  runnable SwcSwitchEventInitRunnable [0.0] {
    symbol "swcSwitchEventInitRunnable"
    modeSwitchEvent exit Mode.currentMode.STARTUP as InitEvent
  }
}

implementation SwcSwitchEventImplementation for SwcSwitchEventType.
    SwcSwitchEventBehavior {
  language c
  codeDescriptor "src"
}
```

A.2 SWC SwcLightRequest

```
package SwcLightRequest

import Interfaces.*
import AUTOSAR.Services.EcuM.EcuM_CurrentMode
import AUTOSAR.Services.EcuM.EcuMFixedTypeMappings

component application SwcLightRequestType {
  ports {
      client OtherLightsSwitchStatusClient requires
    SwitchEventLightRequestCSIf1{
         comSpec readOtherLightsSwitchStatus
      }
      client HeadLightSwitchStatusClient requires
   SwitchEventLightRequestCSIf2{
         comSpec readHeadLightSwitchStatus
      }
      sender beamModeSender provides LightRequestFrontLightManagerSRIf
      sender blinkSender provides LightRequestFrontLightManagerSRIf
      sender lightRequestSender provides LightRequestFrontLightManagerSRIf
      receiver Mode requires EcuM_CurrentMode
    }
}

internalBehavior SwcLightRequestBehavior for SwcLightRequestType {
  dataTypeMappings {
    EcuMFixedTypeMappings
  }

  runnable SwcLightRequestMainRunnable [0.0] {
    symbol "swcLightRequestMainRunnable"
    serverCallPoint synchronous OtherLightsSwitchStatusClient.
   readOtherLightsSwitchStatus
    serverCallPoint synchronous HeadLightSwitchStatusClient.
   readHeadLightSwitchStatus
    dataSendPoint beamModeSender.beamMode
    dataSendPoint blinkSender.blink
    dataSendPoint lightRequestSender.lightRequest
    timingEvent 0.1
  }

  runnable SwcLightRequestInitRunnable [0.0] {
    symbol "SwcLightRequest_Init"
    modeSwitchEvent exit Mode.currentMode.STARTUP as InitEvent
  }
}

implementation SwcLightRequestImplementation for SwcLightRequestType.
   SwcLightRequestBehavior {
  language c
  codeDescriptor "src"
}
```

A.3 SWC FrontLightManager

```
package SwcFrontLightManager

import Interfaces.*
import Prop_Generic.Services.IoHwAb.*
import Prop_Generic.Services.BswM.*
import ArcCore.Services.IoHwAb.*
import AUTOSAR.Services.EcuM.*
import AUTOSAR.Services.Det.*
import AUTOSAR.Services.Dlt.*
import AUTOSAR.Services.Dem.*
import AUTOSAR.Services.WdgM.*

component application SwcFrontLightManagerType{
  ports{
    receiver beamModeReceiver requires LightRequestFrontLightManagerSRIf
    receiver blinkReceiver requires LightRequestFrontLightManagerSRIf
    receiver lightRequestReceiver requires LightRequestFrontLightManagerSRIf
    client ParkingLightClient requires DigitalServiceWrite
    client LeftIndicatorClient requires DigitalServiceWrite
    client RightIndicatorClient requires DigitalServiceWrite
    sender lightBrightnessSender provides FrontLightManagerHeadLightSRIf
    receiver Mode requires EcuM_CurrentMode
    client RunControl requires EcuM_StateRequest
    client Det requires DETService
    client Dlt requires DLTService
    client Dem_TestEvent requires DiagnosticMonitor
    client WdgM_AliveSup requires WdgM_AliveSupervision
    receiver WdgM_LocalMode requires WdgM_IndividualMode
    sender WdgM_StateReq provides WdgMModeRequestInterface
  }
}

internalBehavior SwcFrontLightManagerBehavior for SwcFrontLightManagerType {
  dataTypeMappings {
    EcuMFixedTypeMappings
    WdgMTypeMappings
  }

  runnable SwcFrontLightManagerMainRunnable [0.0] {
    symbol "swcFrontLightManagerMainRunnable"
    dataReceivePoint beamModeReceiver.beamMode
    dataReceivePoint blinkReceiver.blink
    dataReceivePoint lightRequestReceiver.lightRequest
```

```
      server CallPoint synchronous ParkingLightClient.Write
      server CallPoint synchronous LeftIndicatorClient.Write
      server CallPoint synchronous RightIndicatorClient.Write
      data SendPoint lightBrightnessSender.*
      server CallPoint synchronous Det.ReportError
      server CallPoint synchronous Dlt.SendLogMessage
      server CallPoint synchronous Dem_TestEvent.SetEventStatus
      server CallPoint synchronous WdgM_AliveSup.UpdateAliveCounter
      server CallPoint synchronous RunControl.ReleaseRUN
      modeSwitchEvent entry WdgM_LocalMode.currentMode.SUPERVISION_FAILED as
     localModeEvent
      data WriteAccess WdgM_StateReq.*
      timingEvent 0.1
   }

   runnable SwcFrontLightManagerInitRunnable [0.0] {
      symbol "swcFrontLightManagerInitRunnable"
      server CallPoint synchronous RunControl.RequestRUN
      modeSwitchEvent exit Mode.currentMode.STARTUP as InitEvent
   }
}

implementation SwcFrontLightManagerImplementation for
    SwcFrontLightManagerType.SwcFrontLightManagerBehavior {
   language c
   codeDescriptor "src"
}
```

A.4 SWC Headlight

```
package SwcHeadlight

import Interfaces.*
import AUTOSAR.Services.EcuM.EcuM_CurrentMode
import AUTOSAR.Services.EcuM.EcuMFixedTypeMappings
import ArcCore.Services.IoHwAb.*
import Prop_Generic.Services.IoHwAb.*

component application SwcHeadlightType{
   ports{
      receiver lightBrightnessReceiver requires FrontLightManagerHeadLightSRIf
```

```
    client HeadlightPwmDuty requires PwmServiceSetDuty
    receiver Mode requires EcuM_CurrentMode
  }
}

internalBehavior SwcHeadlightBehavior for SwcHeadlightType {
  dataTypeMappings {
    EcuMFixedTypeMappings
  }

  runnable SwcHeadlightMainRunnable [0.0] {
    symbol "swcHeadlightMainRunnable"
    dataReceivePoint lightBrightnessReceiver.*
    serverCallPoint synchronous HeadlightPwmDuty.Set as
   swcHeadlightCallPoint
    timingEvent 0.1
  }

  runnable SwcHeadlightInitRunnable [0.0] {
    symbol "swcHeadlightInitRunnable"
    modeSwitchEvent exit Mode.currentMode.STARTUP as InitEvent
  }
}

implementation SwcHeadlightImplementation for SwcHeadlightType.
   SwcHeadlightBehavior {
  language c
  codeDescriptor "src"
}
```

A.5 Interface Description

This contains only the interfaces used by SWCs i.e. so system provided service's interface is included here.

```
package Interfaces

import ArcCore.Platform.ImplementationDataTypes.*

interface clientServer service SwitchEventLightRequestCSIf1{
  error error_CanNotRead 1
  operation readOtherLightsSwitchStatus possibleErrors error_CanNotRead{
    out uint32 otherLightsSwitchStatus
  }
}

interface clientServer service SwitchEventLightRequestCSIf2{
  error error_CanNotRead 1
  operation readHeadLightSwitchStatus possibleErrors error_CanNotRead{
    out uint8 headLightSwitchStatus
  }
}

record impl LightRequestRecord {
  uint8 off,
  uint8 leftIndicator,
  uint8 rightIndicator,
  uint8 parking
}

interface senderReceiver LightRequestFrontLightManagerSRIf{
  data uint32 beamMode
  data sint8 blink
  data LightRequestRecord lightRequest
}

interface senderReceiver FrontLightManagerHeadLightSRIf{
  data float64 lightBrightness
}
```

A.6 I/O Hardware Abstraction Description

```
package STM3210CEcu.Services.IoHwAb

import ArcCore.Services.IoHwAb.*
import ArcCore.Platform.ImplementationDataTypes.*

component ecuAbstraction IoHwAb {
    ports {
    server Digital_DigitalSignal_OtherLightsSwitchStatus provides
   DigitalServiceRead
```

```
    server Digital_DigitalSignal_HeadLightSwitchStatus provides
   DigitalServiceRead
    server Digital_DigitalSignal_ParkingLight provides DigitalServiceWrite
    server Digital_DigitalSignal_LeftIndicator provides DigitalServiceWrite
    server Digital_DigitalSignal_RightIndicator provides DigitalServiceWrite
    server Pwm_PwmSignal_LED3Duty provides PwmServiceSetDuty
    }
}

internalBehavior IoHwAbBehavior for IoHwAb {
    runnable concurrent DigitalRead [0.0] {
      symbol "IoHwAb_Digital_Read"
        operationInvokedEvent Digital_DigitalSignal_OtherLightsSwitchStatus.
   Read
        operationInvokedEvent Digital_DigitalSignal_HeadLightSwitchStatus.
   Read
    }
    runnable concurrent DigitalWrite [0.0] {
      symbol "IoHwAb_Digital_Write"
        operationInvokedEvent Digital_DigitalSignal_ParkingLight.Write
        operationInvokedEvent Digital_DigitalSignal_LeftIndicator.Write
        operationInvokedEvent Digital_DigitalSignal_RightIndicator.Write
    }
    runnable concurrent PwmSetDuty [0.0] {
        symbol "IoHwAb_Pwm_Set_Duty"
        operationInvokedEvent Pwm_PwmSignal_LED3Duty.Set
    }
    portAPIOption Digital_DigitalSignal_OtherLightsSwitchStatus{
   IoHwAb_SignalType_ 0}
    portAPIOption Digital_DigitalSignal_HeadLightSwitchStatus{
   IoHwAb_SignalType_ 1}
    portAPIOption Digital_DigitalSignal_ParkingLight{IoHwAb_SignalType_ 2}
    portAPIOption Digital_DigitalSignal_LeftIndicator{IoHwAb_SignalType_ 3}
    portAPIOption Digital_DigitalSignal_RightIndicator{IoHwAb_SignalType_ 4}
    portAPIOption Pwm_PwmSignal_LED3Duty{IoHwAb_SignalType_ 0}
}

implementation IoHwAbImpl for IoHwAb.IoHwAbBehavior {
    language c
    codeDescriptor "src"
  vendorId 60
}
```

A.7 ECU Description

```
package Prop

import SwcSwitchEvent.SwcSwitchEventType
import SwcLightRequest.SwcLightRequestType
import SwcFrontLightManager.SwcFrontLightManagerType
import SwcHeadlight.SwcHeadlightType
import STM3210CEcu.Services.IoHwAb.*
import Prop_Generic.Services.EcuM.*
import Prop_Generic.Services.ComM.*
import Prop_Generic.Services.Dcm.Dcm
import Prop_Generic.Services.Det.Det
import Prop_Generic.Services.Dlt.Dlt
import Prop_Generic.Services.Dem.Dem
import Prop_Generic.Services.NvM.NvM
import Prop_Generic.Services.BswM.BswM
import Prop_Generic.Services.WdgM.WdgM


composition TopLevelComposition {
  prototype SwcSwitchEventType swcSwitchEvent
  prototype SwcLightRequestType swcLightRequest
  prototype SwcFrontLightManagerType swcFrontLightManager
  prototype SwcHeadlightType swcHeadlight
  prototype EcuM ecuM
  prototype IoHwAb ioHwAb
  prototype ComM comM
  prototype Dcm dcm
  prototype Det det
  prototype Dlt dlt
  prototype Dem dem
  prototype BswM bswm
  prototype WdgM wdgm
  prototype NvM nvm

  connect ioHwAb.Digital_DigitalSignal_OtherLightsSwitchStatus to
    swcSwitchEvent.OtherLightsSwitchStatusClient
  connect ioHwAb.Digital_DigitalSignal_HeadLightSwitchStatus to
    swcSwitchEvent.HeadLightSwitchStatusClient
  connect ioHwAb.Digital_DigitalSignal_ParkingLight to swcFrontLightManager.
    ParkingLightClient
  connect ioHwAb.Digital_DigitalSignal_LeftIndicator to swcFrontLightManager
    .LeftIndicatorClient
  connect ioHwAb.Digital_DigitalSignal_RightIndicator to
    swcFrontLightManager.RightIndicatorClient
  connect ioHwAb.Pwm_PwmSignal_LED3Duty to swcHeadlight.HeadlightPwmDuty

  connect swcSwitchEvent.OtherLightsSwitchStatusServer to swcLightRequest.
    OtherLightsSwitchStatusClient
  connect swcSwitchEvent.HeadLightSwitchStatusServer to swcLightRequest.
    HeadLightSwitchStatusClient

  connect swcLightRequest.beamModeSender to  swcFrontLightManager.
    beamModeReceiver
  connect swcLightRequest.blinkSender to  swcFrontLightManager.blinkReceiver
  connect swcLightRequest.lightRequestSender to  swcFrontLightManager.
    lightRequestReceiver

  connect swcFrontLightManager.lightBrightnessSender to swcHeadlight.
    lightBrightnessReceiver

  connect ecuM.SR_PropUser to swcFrontLightManager.RunControl
  connect ecuM.currentMode to swcFrontLightManager.Mode
  connect ecuM.currentMode to swcSwitchEvent.Mode
  connect ecuM.currentMode to swcLightRequest.Mode
  connect ecuM.currentMode to swcHeadlight.Mode
  connect det.DS_DetPortReader to swcFrontLightManager.Det
  connect dlt.Dlt_service to swcFrontLightManager.Dlt
  connect dem.Event_TestEvent to swcFrontLightManager.Dem_TestEvent
  connect wdgm.alive_Supervised100msTask to swcFrontLightManager.
    WdgM_AliveSup
  connect wdgm.mode_Supervised100msTask to swcFrontLightManager.
    WdgM_LocalMode
  connect swcFrontLightManager.WdgM_StateReq to bswm.
    modeRequestPort_WdgMMode
}
```

A.8 System Description

```
package Prop


import Prop.TopLevelComposition.*
import Prop.Communication.*
import STM3210CEcu.Services.IoHwAb.*
import Prop_Generic.Services.WdgM.WdgMImpl
import Prop_Generic.Services.ComM.ComMImpl
import Prop_Generic.Services.Dcm.DcmImpl
import Prop_Generic.Services.Dem.DemImpl
import Prop_Generic.Services.Det.DetImpl
import Prop_Generic.Services.Dlt.DltImpl
import Prop_Generic.Services.BswM.BswMComponentImpl
import Prop_Generic.Services.EcuM.EcuMFixedImpl
import Prop_Generic.Services.IoHwAb.IoHwAbImpl
import Prop_Generic.Services.NvM.NvMImpl

import SwcSwitchEvent.SwcSwitchEventImplementation
import SwcLightRequest.SwcLightRequestImplementation
import SwcFrontLightManager.SwcFrontLightManagerImplementation
import SwcHeadlight.SwcHeadlightImplementation

import Interfaces.MemoryCmdIf.command
import Interfaces.MemoryCmdIf.blockId
import Interfaces.MemoryCmdIf.payload
import Interfaces.SwitchEventLightRequestCSIf.switchStatus
import Interfaces.LightRequestFrontLightManagerSRIf.ev
import Interfaces.FrontLightManagerHeadLightSRIf.*


system Prop {
  rootComposition TopLevelComposition

  mapping {
    implMap SwcSwitchEventImplementation to swcSwitchEvent
    implMap SwcLightRequestImplementation to swcLightRequest
    implMap SwcFrontLightManagerImplementation to swcFrontLightManager
    implMap SwcHeadlightImplementation to swcHeadlight
    implMap IoHwAbImpl to ioHwAb

    implMap ComMImpl to comM as comMMapping
    implMap DcmImpl to dcm as dcmMapping
    implMap DemImpl to dem as demMapping
    implMap DetImpl to det as detMapping
    implMap DltImpl to dlt as dltMapping
    implMap NvMImpl to nvm as nvmMapping
    implMap BswMComponentImpl to bswm as bswMMapping
    implMap EcuMFixedImpl to ecuM as ecuMMapping
    implMap WdgMImpl to wdgm as wdgMMapping
  }
}
```

Appendix B

MAST model of Front Light Management

```

Model (
   Model_Name              => FrontLightManager,
   Model_Date              => 2016-05-27,
   System_Pip_Behaviour  => STRICT);

Processing_Resource(
  Type => Regular_Processor,
  Name =>cpu_1,
  Max_Interrupt_Priority => 300,
  Min_Interrupt_Priority => 250,
  Worst_ISR_Switch => 256,
  Avg_ISR_Switch => 256,
  Best_ISR_Switch => 256,
  System_Timer =>
  (Type => Ticker,
      Worst_Overhead => 3938,
      Avg_Overhead => 1191,
      Best_Overhead => 943,
      Period => 72000),
   Speed_Factor => 1.0);

Scheduler (
   Type              => Primary_Scheduler,
   Name              => scheduler_1,
   Host              => cpu_1,
   Policy            =>
      ( Type                    => Fixed_Priority,
        Worst_Context_Switch => 4559.0,
        Avg_Context_Switch   => 4514.0,
        Best_Context_Switch  => 4478.0,
        Max_Priority          => 300,
        Min_Priority          => 1));

    Scheduling_Server (
   Type                          => Regular,
   Name                          => task_light_request,
   Server_Sched_Parameters      =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 8,
        Preassigned  => NO),
   Scheduler                     => scheduler_1);

Scheduling_Server (
   Type                          => Regular,
   Name                          => task_front_light_manager,
   Server_Sched_Parameters      =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 7,
        Preassigned  => NO),
   Scheduler                     => scheduler_1);

Scheduling_Server (
   Type                          => Regular,
   Name                          => task_headlight,
   Server_Sched_Parameters      =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 6,
        Preassigned  => NO),
   Scheduler                     => scheduler_1);

Shared_Resource (
   Type               => immediate_Ceiling_Resource,
   Name               => record_buffer_32,
   Ceiling       => 300,
   Preassigned        => YES);

Shared_Resource (
   Type               => immediate_Ceiling_Resource,
   Name               => buffer_64,
   Ceiling       => 300,
   Preassigned       => YES);

Operation (
   Type                         => Simple,
   Name                         =>
    communication_explicit_less_than_word_size_only_buffer ,
   Worst_Case_Execution_Time  => 23.00,
   Avg_Case_Execution_Time    => 19.00,
   Best_Case_Execution_Time   => 18.00);

Operation (
   Type                         => Simple,
   Name                         => communication_explicit_runnable_to_buffer ,
   Worst_Case_Execution_Time  => 36.00,
   Avg_Case_Execution_Time    => 35.00,
   Best_Case_Execution_Time   => 32.00);

Operation (
   Type                         => Simple,
   Name                         =>
    communication_explicit_only_record_buffer_32_rw ,
   Worst_Case_Execution_Time  => 113.00,
   Avg_Case_Execution_Time    => 109.00,
   Best_Case_Execution_Time   => 108.00,
   Shared_Resources_To_Lock   =>
      ( record_buffer_32),
   Shared_Resources_To_Unlock =>
      ( record_buffer_32));

Operation (
   Type                         => Simple,
   Name                         => communication_explicit_buffer_to_runnable ,
   Worst_Case_Execution_Time  => 41.00,
   Avg_Case_Execution_Time    => 39.00,
   Best_Case_Execution_Time   => 36.00);

Operation (
   Type                       => Composite,
   Name                       => communication_explicit_record_buffer_32_rw ,
   Composite_Operation_List =>
      ( communication_explicit_runnable_to_buffer ,
        communication_explicit_only_record_buffer_32_rw ,
        communication_explicit_buffer_to_runnable));

```

```
Operation (
   Type                         => Composite,
   Name                         => communication_explicit_less_than_word_size,
   Composite_Operation_List =>
      ( communication_explicit_runnable_to_buffer,
        communication_explicit_less_than_word_size_only_buffer,
        communication_explicit_buffer_to_runnable));

Operation (
   Type                          => Simple,
   Name                          => communication_explicit_only_buffer_64_rw,
   Worst_Case_Execution_Time   => 108.00,
   Avg_Case_Execution_Time     => 106.00,
   Best_Case_Execution_Time    => 104.00,
   Shared_Resources_To_Lock    =>
      ( buffer_64),
   Shared_Resources_To_Unlock =>
      ( buffer_64));

Operation (
   Type                         => Composite,
   Name                         => communication_explicit_buffer_64_rw,
   Composite_Operation_List =>
      ( communication_explicit_runnable_to_buffer,
        communication_explicit_only_buffer_64_rw,
        communication_explicit_buffer_to_runnable));

Operation (
   Type                          => Simple,
   Name                          =>
    start_runnable_non_implicit_from_event_occurred_to_runnable,
   Worst_Case_Execution_Time   => 306.00,
   Avg_Case_Execution_Time     => 304.00,
   Best_Case_Execution_Time    => 302.00);

Operation (
   Type                          => Simple,
   Name                          =>
    end_runnable_non_implicit_from_runnable_to_again_going_back_to_wait,
   Worst_Case_Execution_Time   => 41.00,
   Avg_Case_Execution_Time     => 37.00,
   Best_Case_Execution_Time    => 36.00);

Operation (
   Type                          => Simple,
   Name                          => start_runnable_synch_server ,
   Worst_Case_Execution_Time     => 77.00 ,
   Avg_Case_Execution_Time       => 76.00 ,
   Best_Case_Execution_Time      => 72.00) ;

Operation (
   Type                          => Simple ,
   Name                          => end_runnable_synch_server ,
   Worst_Case_Execution_Time     => 81.00 ,
   Avg_Case_Execution_Time       => 81.00 ,
   Best_Case_Execution_Time      => 81.00) ;

Operation (
   Type                          => Simple ,
   Name                          => iohwab_digital_write ,
   Worst_Case_Execution_Time     => 1296.0 ,
   Avg_Case_Execution_Time       => 1262.0 ,
   Best_Case_Execution_Time      => 1202.0) ;

Operation (
   Type                          => Simple ,
   Name                          => iohwab_digital_read ,
   Worst_Case_Execution_Time     => 644.00 ,
   Avg_Case_Execution_Time       => 615.00 ,
   Best_Case_Execution_Time      => 590.00) ;

Operation (
   Type                          => Simple ,
   Name                          => iohwab_pwm_write ,
   Worst_Case_Execution_Time     => 477.00 ,
   Avg_Case_Execution_Time       => 473.00 ,
   Best_Case_Execution_Time      => 468.00) ;

Operation (
   Type                          => Simple ,
   Name                          => init_runnables_only_runnable_time ,
   Worst_Case_Execution_Time     => 3640 ,
   Avg_Case_Execution_Time       => 3650 ,
   Best_Case_Execution_Time      => 3690) ;

Operation (
   Type                       => Composite ,
   Name                       => init_runnables ,
   Composite_Operation_List   =>
      ( start_runnable_non_implicit_from_event_occurred_to_runnable ,
       init_runnables_only_runnable_time ,
       end_runnable_non_implicit_from_runnable_to_again_going_back_to_wait)
   );

Operation (
   Type                            => Simple ,
   Name                            =>
    swcheadlightswitchstatusrunnable_from_begining_to_headlightswitchstatusclient_read
    ,
   Worst_Case_Execution_Time   => 1400.00,
   Avg_Case_Execution_Time     => 1200.00,
   Best_Case_Execution_Time    => 900.00);

Operation (
   Type                            => Simple ,
   Name                            =>
    swcheadlightswitchstatusrunnable_from_headlightswitchstatusclient_read_to_end
    ,
   Worst_Case_Execution_Time   => 2300.00,
   Avg_Case_Execution_Time     => 2000.00,
   Best_Case_Execution_Time    => 1800.00);

Operation (
   Type                        => Composite ,
   Name                        => swcheadlightswitchstatusrunnable ,
   Composite_Operation_List =>
      (
    swcheadlightswitchstatusrunnable_from_begining_to_headlightswitchstatusclient_read
    ,
        iohwab_digital_read ,

    swcheadlightswitchstatusrunnable_from_headlightswitchstatusclient_read_to_end
    ));

Operation (
   Type                            => Simple ,
   Name                            =>
    swcotherlightsswitchstatusrunnable_from_begining_to_headlightswitchstatusclient_read
    ,
   Worst_Case_Execution_Time   => 1400.00,
   Avg_Case_Execution_Time     => 1200.00,
   Best_Case_Execution_Time    => 900.00);

Operation (
   Type                            => Simple ,
   Name                             =>
    swcotherlightsswitchstatusrunnable_from_headlightswitchstatusclient_read_to_end
    ,
   Worst_Case_Execution_Time   => 2300.00,
   Avg_Case_Execution_Time     => 2200.00,
   Best_Case_Execution_Time    => 1800.00);

Operation (
   Type                        => Composite,
   Name                        => swcotherlightsswitchstatusrunnable,
   Composite_Operation_List =>
      (
    swcotherlightsswitchstatusrunnable_from_begining_to_headlightswitchstatusclient_read
    ,
        iohwab_digital_read,

    swcotherlightsswitchstatusrunnable_from_headlightswitchstatusclient_read_to_end
    ));

Operation (
   Type                             => Simple,
   Name                             =>
    swclightrequestmainrunnable_from_begining_to_readotherlightsswitchstatus,
   Worst_Case_Execution_Time   => 8100.00,
   Avg_Case_Execution_Time     => 7800.00,
   Best_Case_Execution_Time    => 7700.00);

Operation (
   Type                             => Simple,
   Name                             =>
    swclightrequestmainrunnable_from_readotherlightsswitchstatus_to_lightrequest
    ,
   Worst_Case_Execution_Time   => 14000.00,
   Avg_Case_Execution_Time     => 9900.00,
   Best_Case_Execution_Time    => 9000.00);

Operation (
   Type                             => Simple,
   Name                             =>
    swclightrequestmainrunnable_from_lightrequest_to_readheadlightswitchstatus
    ,
   Worst_Case_Execution_Time   => 900.00,
   Avg_Case_Execution_Time     => 900.00,
   Best_Case_Execution_Time    => 500.00);

```

```
Operation (
    Type                         => Simple,
    Name                         =>
     swclightrequestmainrunnable_from_readheadlightswitchstatus_to_beammode,
    Worst_Case_Execution_Time    => 1400.00,
    Avg_Case_Execution_Time      => 1300.00,
    Best_Case_Execution_Time     => 900.00);

Operation (
    Type                         => Simple,
    Name                         =>
     swclightrequestmainrunnable_from_beammode_to_blink,
    Worst_Case_Execution_Time    => 9.00,
    Avg_Case_Execution_Time      => 9.00,
    Best_Case_Execution_Time     => 5.00);

Operation (
    Type                         => Simple,
    Name                         =>
     swclightrequestmainrunnable_from_blink_till_end,
    Worst_Case_Execution_Time    => 900.00,
    Avg_Case_Execution_Time      => 900.00,
    Best_Case_Execution_Time     => 500.00);

Operation (
    Type                       => Composite,
    Name                       => swclightrequestmainrunnable,
    Composite_Operation_List =>
        (
     swclightrequestmainrunnable_from_begining_to_readotherlightsswitchstatus,
          start_runnable_synch_server,
          swcotherlightsswitchstatusrunnable,
          end_runnable_synch_server,

     swclightrequestmainrunnable_from_readotherlightsswitchstatus_to_lightrequest
     ,
          communication_explicit_record_buffer_32_rw,

     swclightrequestmainrunnable_from_lightrequest_to_readheadlightswitchstatus
     ,
          start_runnable_synch_server,
          swcheadlightswitchstatusrunnable,
          end_runnable_synch_server,

     swclightrequestmainrunnable_from_readheadlightswitchstatus_to_beammode,
          communication_explicit_less_than_word_size ,
          swclightrequestmainrunnable_from_beammode_to_blink ,
          communication_explicit_less_than_word_size ,
          swclightrequestmainrunnable_from_blink_till_end));

Operation (
    Type                           => Composite ,
    Name                           => swclightrequest ,
    Composite_Operation_List =>
        ( start_runnable_non_implicit_from_event_occurred_to_runnable ,
          swclightrequestmainrunnable ,
          end_runnable_non_implicit_from_runnable_to_again_going_back_to_wait)
     );

Operation (
    Type                             => Simple ,
    Name                             =>
     swcfrontlightmanagermainrunnable_from_beginging_to_beam_mode ,
    Worst_Case_Execution_Time   => 1400.00 ,
    Avg_Case_Execution_Time     => 1300.00 ,
    Best_Case_Execution_Time    => 900.00);

Operation (
    Type                             => Simple ,
    Name                             =>
     swcfrontlightmanagermainrunnable_from_beam_mode_to_light_brightness ,
    Worst_Case_Execution_Time   => 1800.00 ,
    Avg_Case_Execution_Time     => 1500.00 ,
    Best_Case_Execution_Time    => 1400.00);

Operation (
    Type                             => Simple ,
    Name                             =>
     swcfrontlightmanagermainrunnable_from_light_brightness_to_light_request ,
    Worst_Case_Execution_Time   => 90000.00 ,
    Avg_Case_Execution_Time     => 80000.00 ,
    Best_Case_Execution_Time    => 50000.00);

Operation (
    Type                             => Simple ,
    Name                             =>
     swcfrontlightmanagermainrunnable_from_light_request_to_blink ,
    Worst_Case_Execution_Time   => 413100.0 ,
    Avg_Case_Execution_Time     => 340200.0 ,
    Best_Case_Execution_Time    => 136800.0);
```

```
Operation (
    Type                          => Simple,
    Name                          =>
     swcfrontlightmanagermainrunnable_from_blink_till_end,
    Worst_Case_Execution_Time   => 41270.0,
    Avg_Case_Execution_Time     => 41010.0,
    Best_Case_Execution_Time    => 40950.0);

Operation (
    Type                        => Composite,
    Name                        => swcfrontlightmanagermainrunnable,
    Composite_Operation_List  =>
        ( swcfrontlightmanagermainrunnable_from_beginging_to_beam_mode,
          communication_explicit_less_than_word_size,
          swcfrontlightmanagermainrunnable_from_beam_mode_to_light_brightness,
          communication_explicit_buffer_64_rw,

     swcfrontlightmanagermainrunnable_from_light_brightness_to_light_request,
          communication_explicit_record_buffer_32_rw,
          swcfrontlightmanagermainrunnable_from_light_request_to_blink,
          communication_explicit_less_than_word_size,
          swcfrontlightmanagermainrunnable_from_blink_till_end));

Operation (
    Type                        => Composite,
    Name                        => swcfrontlightmanager,
    Composite_Operation_List  =>
        ( start_runnable_non_implicit_from_event_occurred_to_runnable,
          swcfrontlightmanagermainrunnable,
          end_runnable_non_implicit_from_runnable_to_again_going_back_to_wait)
     );

Operation (
    Type                          => Simple,
    Name                          =>
     swcheadlightmainrunnable_from_beginging_to_light_brightness,
    Worst_Case_Execution_Time   => 2300.00,
    Avg_Case_Execution_Time     => 2000.00,
    Best_Case_Execution_Time    => 1800.00);

Operation (
    Type                          => Simple,
    Name                          =>
     swcheadlightmainrunnable_from_light_brightness_to_pwm_duty_set,
    Worst_Case_Execution_Time   => 87300.00,
    Avg_Case_Execution_Time     => 77100.00,
    Best_Case_Execution_Time    => 68900.00);

Operation (
    Type                        => Simple,
    Name                        =>
     swcheadlightmainrunnable_from_pwm_duty_set_till_end,
    Worst_Case_Execution_Time   => 900.00,
    Avg_Case_Execution_Time     => 900.00,
    Best_Case_Execution_Time    => 500.00);

Operation (
    Type                      => Composite,
    Name                      => swcheadlightmainrunnable,
    Composite_Operation_List  =>
       ( swcheadlightmainrunnable_from_beginging_to_light_brightness,
         communication_explicit_buffer_64_rw,
         swcheadlightmainrunnable_from_light_brightness_to_pwm_duty_set,
         iohwab_pwm_write,
         swcheadlightmainrunnable_from_pwm_duty_set_till_end));

Operation (
    Type                      => Composite,
    Name                      => swcheadlight,
    Composite_Operation_List  =>
       ( start_runnable_non_implicit_from_event_occurred_to_runnable,
         swcheadlightmainrunnable,
         end_runnable_non_implicit_from_runnable_to_again_going_back_to_wait)
    );

Transaction (
    Type            => regular,
    Name            => transaction_init_all_runnables,
    External_Events =>
       ( ( Type  => Singular,
           Name  => event_singular_init_runnale,
           Phase => 0.000)),
    Internal_Events =>
       ( ( Type => Regular,
           Name => internal_event)),
    Event_Handlers  =>
       ( (Type                => Activity,
          Input_Event         => event_singular_init_runnale,
          Output_Event        => internal_event,
          Activity_Operation => init_runnables,
          Activity_Server    => task_front_light_manager)));

Transaction (
    Type             => regular,
    Name             => transaction_light_request,
    External_Events =>
       ( ( Type        => Periodic,
           Name        => event_periodic_lr_50ms,
           Period      => 720000,
           Max_Jitter => 0.000,
           Phase       => 0.000)),
    Internal_Events =>
       ( ( Type => Regular,
           Name => internal_event)),
    Event_Handlers  =>
       ( (Type                  => System_Timed_Activity,
          Input_Event           => event_periodic_lr_50ms,
          Output_Event          => internal_event,
          Activity_Operation => swclightrequest,
          Activity_Server    => task_light_request)));

Transaction (
    Type             => regular,
    Name             => transaction_front_light_manager,
    External_Events =>
       ( ( Type        => Periodic,
           Name        => event_periodic_flm_50ms,
           Period      => 720000,
           Max_Jitter => 0.000,
           Phase       => 0.000)),
    Internal_Events =>
       ( ( Type => Regular,
           Name => internal_event)),
    Event_Handlers  =>
       ( (Type                  => System_Timed_Activity,
          Input_Event           => event_periodic_flm_50ms,
          Output_Event          => internal_event,
          Activity_Operation => swcfrontlightmanager,
          Activity_Server    => task_front_light_manager)));

Transaction (
    Type             => regular,
    Name             => transaction_headlight,
    External_Events =>
```

```
      ( ( Type        => Periodic ,
          Name        => event_periodic_hl_50ms ,
          Period      => 720000   ,
          Max_Jitter  => 0.000 ,
          Phase       => 0.000)) ,
   Internal_Events =>
      ( ( Type => Regular ,
          Name => internal_event)) ,
   Event_Handlers  =>
      ( (Type                  => System_Timed_Activity ,
         Input_Event           => event_periodic_hl_50ms ,
         Output_Event          => internal_event ,
         Activity_Operation    => swcheadlight ,
         Activity_Server       => task_headlight)));
```