CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Logging technologies in software design for real-time systems and simulators

Master's thesis in Embedded Electronic System Design

LIKAI CHU

NHAT NGUYEN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Logging technologies in software design for real-time systems and simulators

LIKAI CHU
NHAT NGUYEN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Logging technologies in software design for real-time systems and simulators
LIKAI CHU
NHAT NGUYEN

Industrial Supervisor: Henrik Lönn, Volvo Group Trucks Technology
Academic Supervisor: Jan-Philipp Steghöfer, Department of Computer Science and Engineering
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Logging technologies in software design for real-time systems and simulators
LIKAI CHU, NHAT NGUYEN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The aim of this work was to study and develop logging technologies for a software simulator (ADAPT) and corresponding embedded system on trucks. One of the targets of our thesis is developing a logging technology for the ADAPT simulation environment. Another target is to build up a proof of concept that a logging software can operate on the real-time systems of trucks.

Both software simulator and embedded system on trucks generate data during operations. While these data are beneficial for trucks development, the question of how to log and use the data among the two systems should be researched. Furthermore, how can we evaluate the results for previous question is also important to establish.

In this thesis, iterative design science approach is being used. By applying this method, we are constantly developing the logging system as well as evaluating our design. At the same time, requirements and knowledge base are updated to adapt with the current progress.

As a contribution of this thesis, a functional logging technology is developed for the ADAPT system, and the concept of building functional logging software on the real-time system of trucks is proved.

Keywords: ADAPT Simulator, Logger, Replayer, WCET, Schedulability, real-time system, AUTOSAR.

Acknowledgements

We would like say thank you to our supervisors, Jan-Philipp and Henrik, for their dedicated supports and patience, a lot of patience with us during this thesis work. We are also thankful to Vector and aiT companies who kindly support us with their tools, and of course, the friendly, humorous, generous employee in Volvo GTT.

Likai Chu and Nhat Nguyen, Gothenburg, May 2020

Abbreviation

ASAM Standardization for Automotive Development
AUTOSAR Automotive Open System Architecture
BCET Best Case Execution Time
CAN Controller Area Network
CLI Command Line Interface
ECU Electronic Control Unit
EDF Early Deadline First
EE Electric and Electronic
FMU Functional Movkup Unit
IO Input and Output
LIN Local Interconnect Network
ROS Robot Operating System
RTE Runtime Environment
SWC Software Component
TAT Turn-Around-Time
VSB Virtual Signal Bus

Contents

List of Figures

List of Tables

1
Introduction

In recent years, logged data becomes increasingly important for truck research and development process because logged data can contribute to tracking and monitoring of the trucks, function verification and machine learning for autonomous driving. However, the process of logging the truck data is quite challenging.

First of all, the amount of data from the trucks nowadays are huge. There are two reasons for this, firstly trucks are equipped with more Electronic Control Units (ECUs) as well as more sensors than before, both of them produce useful data that needs to be collected. Secondly some sensors, especially cameras and Light detection and ranging (LiDAR) systems, are becoming common in trucks due to their application in active safety and autonomous driving. And these sensors can produce data massively, e.g. the camera may produce tens of megabytes of photo per second. Therefore data collecting requires a process that has high throughput.

Second, the data needs to be arranged in a way that can serve the research and development of the trucks. That is, in order to have the reproducibility for verification and validation, it is usually the case that a specific set of data is needed, e.g. a set of data reflects the engine information including engine speed, temperature, etc. To achieve this, the logging process should be able to be customized so that several types of data can be selected in the same data set. Also, the timing information should be available for all data so this set can be "replayed", i.e. each data value can be output in real time to restore the occurred scenario.

Third, logging data means dealing with different environments. In the context of trucks, different frameworks developed by different suppliers as well as various assets, e.g. software programs running on a simulation environment or ECUs on the truck, require to be handled properly, i.e. a unified logging solution should be provided to cope with various components.

Finally, when data is logged from the ECUs or certain hardware on the truck, the logging process becomes critical to timing because loss of important data can occur by failing the timing constraints or even a systematical crash that the real-time system on the ECU stop working.

By researching and implementing novel data logging technologies, we can provide robust solutions addressing the above logging problems for trucks as well as the simulation environment provided by Volvo Group, which can help to improve quality

of data logging for trucks and contribute to the truck research and development process.

1.1 Problem Formulation

There is a need of logging systems for simulators as well as Electrical and Electronic (EE) systems on trucks. The logged data will be analyzed by many further analyses and development processes. The data should be organized in some structures or formats by design entities. In turn, the designs need evaluating properly to prove for their correctness. Therefore, this thesis seeks systematic solutions by studying the following research and development (R&D) question:

- How can we log data from a target simulator and real-time systems on trucks?
- How can we replay the data from log file to a simulator so that we can mimic its internal components' operation?
- How can we evaluate the logging system corresponding to the design that has been validated the simulator?

1.2 Goals

This thesis project aims to research and design a prototype or framework which can provide a unified logging solution of different platforms such as simulator and EE systems on trucks. The solution will be design to have these characteristics:

- Well-defined and flexible log data structure
- Scalable and extendable configurability
- Robust and efficient log data acquisition
- Proper interfaces with existing system
- Runtime flexibility and integrity

To elaborate, characteristics such as value and unit of the log data are included in the structure and they can be adjusted under different circumstances. The logging system is able to capture data from different parts of the platform and can handle varying size of data sets. Also, the logging process shall be reliable with minimum failure and have little impact on performance of the target platform. Finally, an interface for user to customize shall be provided and a real-time analysis is needed.

1.3 Limitations

An end-to-end logging solution may have extensive design considerations, however, we only focus on collecting, replaying and packaging log data within the objectives in Section 1.2. Other off-board services or concerns will not be part of the central focus of this thesis project including:

- Clock synchronization aspects for timestamping in the truck
- Ethical sensitivity concerns of logged data
- Security services for logged data
- Off-board infrastructure of logged data for access portal services

Besides, there are commercial logger solutions which can operate by using additional devices with certain interfaces to record data from vehicles such as GL Logger [1] from Vector. However, such logger solutions are usually complete external hardware systems or devices connected to vehicles. Therefore, the comparison with other logger devices or systems is limited due to the mismatch of use-cases, environment and design level. We will evaluate our design by using the in-house simulator and real-time analysis framework.

1.4 Report Outline

The remaining parts of this thesis follow a structure with Background, Method, Results and Conclusion. Chapter 2 provides a basic background on the related fields such as the simulation platform, logging format comparison, real-time systems and scheduling, software and tools, etc. which are necessary to understand the contents of the report. In Chapter 3, we describe our research methodology and planning for the whole project. After that, Chapter 4 presents our results and evaluations for our design. This chapter also explains our implementation and our benchmarks used to evaluate the target system. Finally, discussions regarding these results are conveyed in Chapter 5, which are followed by a few concluding remarks.

2
Background

This chapter will present some of project's backgrounds and theories behind them.

2.1 Project Background

2.1.1 Project Targets

The block diagram shown in Figure 2.1 is the logging scenario in the truck development and it is what our logging system design should cope with. In the illustration of the subsystem shown in the right block of the figure, it is the in-house simulator (ADAPT [2]) Inside the simulator, the Virtual Data Bus acts as a virtual backbone network to transfer the data among the different software or emulated hardware modules which all connected with the bus. While the hardware models simulate all non-software components behaviors in the real world, software modules give us the simulated results from ECUs. To realize data acquisition for this simulator, a main agent logger is needed to collect data from both software and hardware modules. Also, the logged data shall be reused through a replayer, i.e. software component for replaying. On the left side of Figure 2.1, the EE system on the trucks is illustrated. It is a real-time system running on different ECUs which connect together using Backbone Networks. While ECUs send their external data to the Backbone Networks and thus the data can be collected from Controller Area Network (CAN) networks [3], they keep their internal data (e.g. program states, intermediate results) separated inside their local memory. It means that the backbone listener cannot collect the ECUs internal data directly.

To summarize, we can divide the data of this scenario into three categories:

- Simulated data from simulator (e.g., mechanical, electrical, hydraulic, etc.);
- ECUs external data;
- ECUs internal data.

In this project, we have access to the mentioned in-house simulator ADAPT but the EE systems shown in Figure 2.1 are not accessible. Considering this, we have the following project targets:

- A logging component and a replaying component operating over ADAPT simulator should be implemented, which means the simulator data from simulator can be logged;
- The replayer should be able to reuse the data from the logging counterpart;

- Both the logger and replayer should meet the characteristics set in the Goals;
- Logging technologies running on truck EE systems should be proved by using hardware simulation tool, the ECU data may not be acquired directly from the ECU.

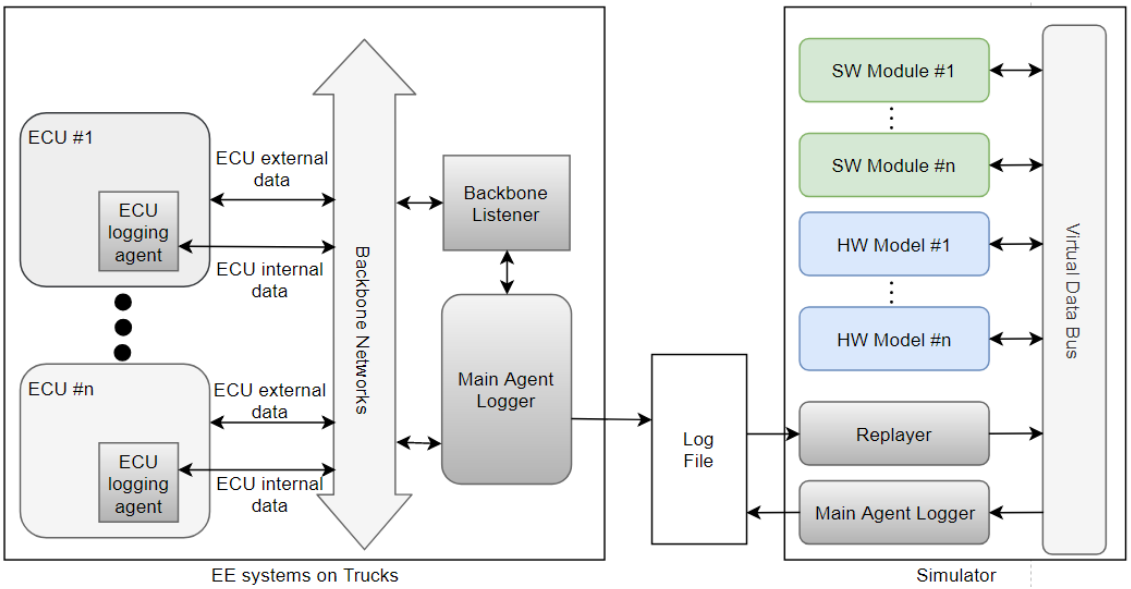


Figure 2.1: A tentative block diagram of our design for two targets: Electrical and Electronic systems on trucks and Simulator

2.1.2 Further details

For achieving the target of implementing the logging component in the ADAPT simulator, we can collect data we need from simulator by reading variables from Virtual Data Bus because the whole simulation runs on the same computer system with the same memory access. For the replayer, when operating, it shall act as any node in the simulator as an open loop. Also, It is worth to note that the logged files can be acquired by prior simulation as well as the real data collected from EE systems on trucks. By using these data for later simulation steps, developers can simulate and analyze more extensively for corner cases of trucks' operation, and thus improve the development process as well as the product.

For the last target of performing the hardware simulation, we are aiming at the scenario of a single ECU logging agent inside an ECU as it is shown in Figure 2.1. The reason that it is hard to perform hardware simulations with an intact EE system containing multiple ECUs is because it requires detailed EE systems information on trucks and may also vary among different truck models. Instead, the hardware simulation is performed based on a single commonly-used automotive ECU. Therefore, the ECU external data mentioned above in Figure 2.1 may be omitted.

Considering this target, the logging agent will be embedded into the system of the existing ECU and may consume ECU computation time or add more delays which

may violate the deadline of other tasks. This problem is very serious because it may cause system crash and hence should be avoided. As a result, real-time analyses and evaluations (e.g., Amalthea [4]) are needed to verify that the logging agent can be applicable. And in our case, our resulting implementation from the ADAPT simulator is considered to be reused upon real-time analyses and evaluations.

2.2 Real-time Systems

As mentioned in the section above, real-time analyses are to be performed as a part of the hardware simulation. The logging process follows characteristics of a real-time system which must meet strict timing constraints, i.e., deadlines, in order to provide a correct service. A real-time system is commonly defined as: "A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated" [5].

In a real-time system, tasks (i.e., pieces of software) can have hard or soft deadlines. While missing a hard deadline in a real-time system can result in physical damage or loss of life in case of automotive system, soft deadlines may be missed without causing a catastrophic failure, but rather result in reduced quality of the provided service. In automotive systems, the important tasks in control systems such as brake-by-wire systems must meet hard deadlines. There are also numerous systems with soft deadlines, e.g., infotainment or multimedia systems which only experience reduced quality of service when their tasks miss the deadlines.

Real-time system tasks can be divided into periodic tasks and sporadic tasks depending on how often they are executed in a system. In this thesis, we focus mainly on periodic tasks since the EE systems on the truck are real-time systems that can be modeled as $\tau_i = \{C_i, T_i, D_i\}$, where C_i, T_i, D_i represent the corresponding Worst-Case Execution Time (WCET), period and deadline of a task τ_i respectively. While the periods and deadlines of tasks come from the specification of the corresponding software specification, the WCET depends heavily on the coding nature of the software implementation on a specific target hardware. Therefore, the WCET of tasks should be estimated beforehand for further analyses.

2.3 Tasks Schedulability

As mentioned above, we need to assure that all tasks in a real-time system will meet their corresponding deadlines. In the process, we will determine whether a task set of a real-time system can be scheduled so that every instance of a task will complete by the time constraints. This process is called schedulability analysis and contains two main aspect which are priority assignment and feasibility test problems. While priority assignment solves the problem of determining which task has higher priority to be executing compared with the others, the latter aims to confirm the correctness

of the result of priority assignments.

One of the earliest studies in this field is the approach from Liu and Layland on sufficient and necessary test using guarantee bound analysis [6]. The authors proposed and proved that we can assign high priority for tasks which have shorter period time, that is called Rate Monotonic (RM). In addition, by taking utilization test as shown below, when the deadline equals the period for each task, we can also confirm that the task set will always meet deadlines.

$$U_{RM}(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1), D_i = T_i$$

where n is the number of task in the task set. This test, however, is limited by a low utilization bound because the absolute bound is $\log(2)$ when $n \to \infty$.

Another approach is shown in [7] which described the assignment by earliest deadline first (EDF) for task instances. The guarantee bound for this method is increased to

$$U_{EDF}(n) \leq 1, D_i = T_i$$

In this thesis, we will use RM and EDF as task schedulability approaches.

2.4 WCET Estimation - Approaches and Tools

For the real-time systems, the WCET is an important factor since the WCET indicates the longest execution time which is requested to meet the timing constraints. So how to find out the WCET for real-time systems? Based on the research in [8], there are two possible ways to determine WCET: dynamic timing analysis and static timing analysis.

2.4.1 Approaches

The tasks performed by the systems can be dependent on a great amount of conditions and the variation of parameters. Therefore, it can be impossible to explore all the possible ways to execute the systems and thus the exact WCET of a certain system is hard to get.

One possible way to calculate the WCET is to use prediction methods(measured-base approach) [8]. In this method, only a part of the total execution times will be generated. The Figure 2.2 below gives a brief explanation of the method. In the Figure 2.2, the white curve depicts the execution times which are observed after running and the dark curve depicts all the possible execution times of the real-time system. Through limited times of execution, both maximal and minimal observed execution time from the result can be determined. Then based on the existing minimal and maximal observed execution time, WCET and BCET (best-case execution time) can be estimated by increasing and decreasing the current observation. So in the Figure 2.2, we can see a relatively lower BCET and a relatively higher WCET.

This method has a drawback. Because the margin between the estimated time and the observed time should be wide enough but also tight, an overestimation or underestimation is still possible.

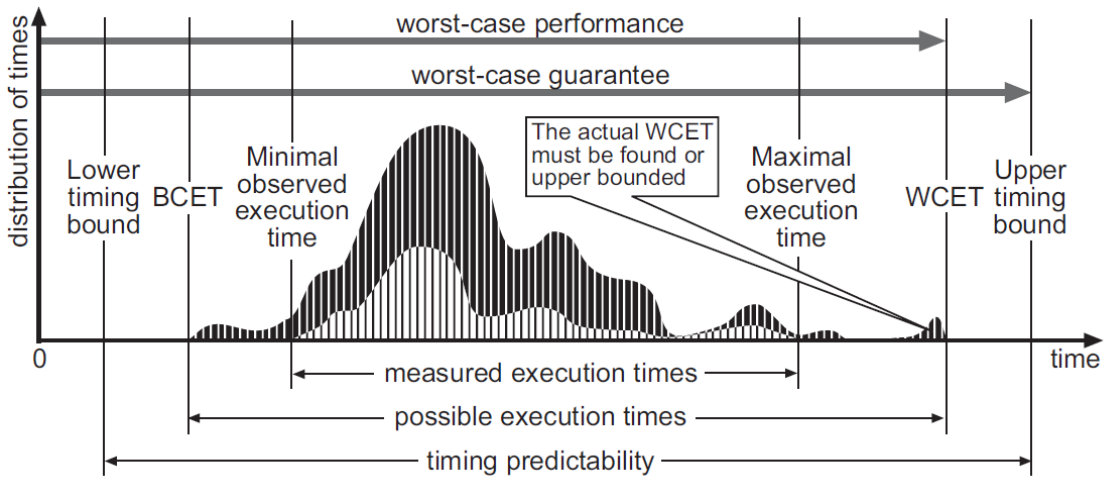


Figure 2.2: In measured-base approach, the execution time varies depending on input data or environment [8].

The second method to estimate the WCET is to examining the software without executing it directly on the hardware. In such an analysis, a software is considered a set of tasks which are either pieces of source code or disassembled executable binaries. By using timing information about the real hardware that the task will execute on, we can acquire an upper bound on the time required to execute a given task on a given hardware platform. This timing bound should be close to the actual WCET. Such methods are referred to as static methods [8]. This means that, using a static method, the obtained upper timing bound is larger than the actual WCET because the timing information from hardware always contain an upper margin. Thus, the upper timing bound is pessimistic (since it is higher than the actual WCET), and should be as close as possible to the actual WCET. In fact, this method requires highly detailed descriptions of the target hardware such as instruction/data caches, branch prediction and instruction pipelines, CPU pipeline and the whole memory hierarchy, etc.

2.4.2 Tools

There are various kinds of WCET estimation tools in the market and the methods they use are largely based on the two methods as described above. Most tools that we investigated are using the static approach. It includes aiT by AbsInt[9], Bound-T by Tidorum[10] and Chronos by National University of Singapore[11]. In our project, you choose the aiT from AbsInt as our WCET estimation tool. It uses the second approach, i.e., static methods, as we have described above, and the following

figure shows the process of how analyses are performed.

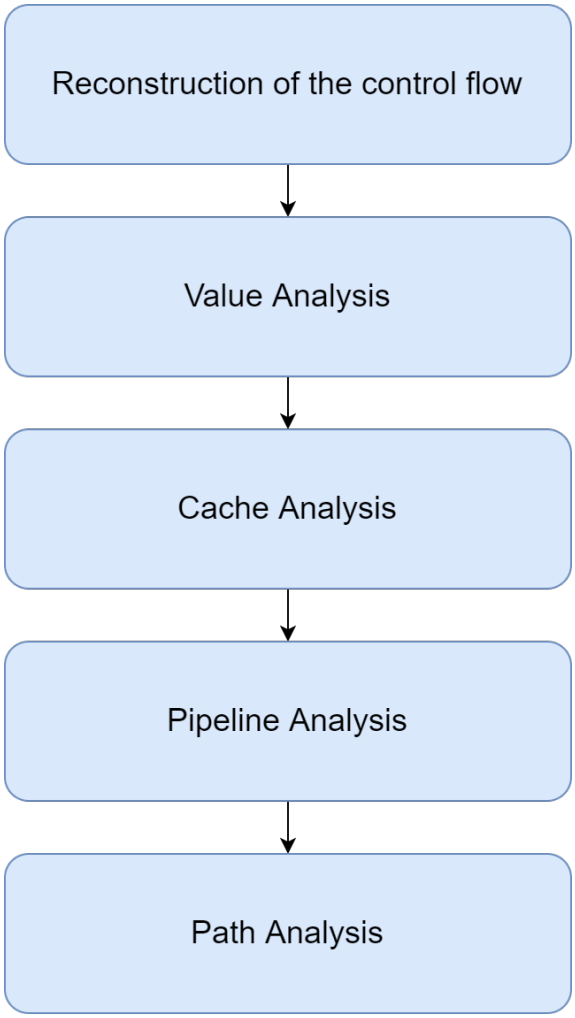


Figure 2.3: AbsInt aiT tool analysis process

In the first stage as in Figure 2.3, the binaries of the programs are used to construct an annotated control flow which shows how definitions, instructions, function calls are arranged within execution as well as other important information required by later stages of analysis. After the construction of the control flow, value analysis is performed. In this stage, value ranges are calculated so that indirect memory accesses can be estimated. After the value analysis, cache analysis is done. It estimates the cache misses and from that the preemption cost of cache can be calculated. Then the pipeline analysis is performed. In this analysis, the behavior of the instruction pipeline is analyzed so that the execution times for sets of sequential instructions are estimated. Finally, in the stage of path analysis, the control flow can be transformed into an integer linear program. And the solution to it reveals the WCET estimation.

2.5 AUTOSAR

In this project, many software and hardware component which we will work with follow AUTomotive Open System ARchitecture (AUTOSAR [12]) standard. AUTOSAR is a standardized automotive software architecture, which also provides a description of development steps that have to be executed during system development. AUTOSAR has defined nine top-level goals: transferability, scalability, broad variety, open architecture, dependable development, sustainable utilization, various partner collaboration, functionality standardization and applicability for international automotive software as well as automotive ECUs [13].

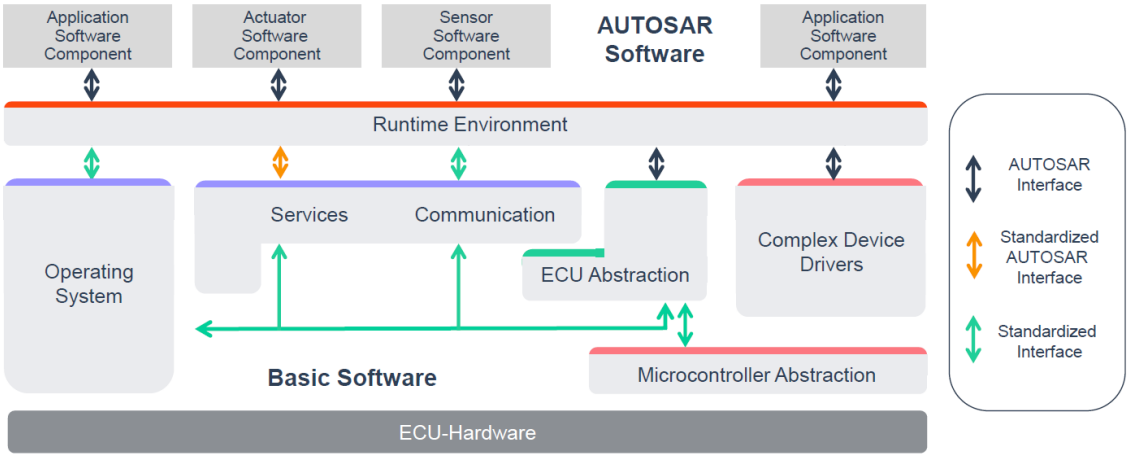


Figure 2.4: AUTOSAR Classic Platform Software Architecture [13].

AUTOSAR today standardizes two different software platforms namely Classic and Adaptive platform [13]. In this project we will focus mainly on Classic platform because our existing systems are using this platform. In Classic AUTOSAR, the application software is divided into different Software Components (SWCs), which in turn consist of runnables [14]. A runnable can be described as a sequence of instructions that can be executed and scheduled independently i.e., an atomic read-modify-write operation. This means that each software component is a set of runnables. An SWC can be viewed as a piece of application software that is independent of the ECU that it is running on. Many services needed by the SWCs are provided by the Run-Time Environment (RTE), which in turn uses the AUTOSAR OS and services for the system, such as memory management and communication frameworks called Basic Software as seen in Figure 2.4, which shows a high-level view of a system running AUTOSAR.

In order to execute properly, the runnables will form tasks in runnable-to-task or partitioning process of AUTOSAR. These tasks define execution units with specific amount of instructions and dependencies which are derived from read and write accesses to memory. Each task can become a large independent partition after this process. The tasks are then assigned in task allocation or mapping process where they are utilized with given resources in form of hardware. For AUTOSAR

applications, both partitioning and mapping process must be considered carefully to search for acceptable solutions which are verified in evaluating or tracing phase.

2.6 Software-in-the-loop Platform (ADAPT)

ADAPT is an integration and simulation framework which has been developed in the HeavyRoad project of Volvo Trucks AB [2, 15]. This framework is managed by the ADAPT consortium and implemented in an in-house developed software of Volvo, which will be used as one of the two main platform targets for Logger and Replayer modules/components in this thesis project.

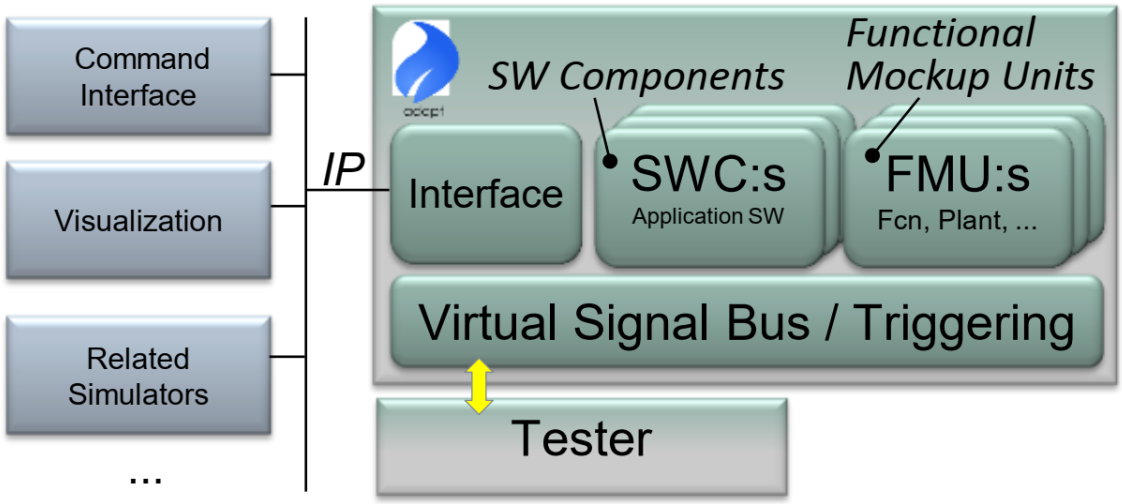


Figure 2.5: The ADAPT system consists of Virtual Signal Bus for triggering components, Interface with other tools or outside environment, SWC and FMU [2].

This platform will play the role of an executor of AUTOSAR SWCs along with preliminary functions, sensors, actuators, plants and environment in forms of models. These components are triggered by the ADAPT framework in the simulation. In the collection phase, they will exchange data over a virtual signal bus, which can be viewed as a generalization of the AUTOSAR virtual function bus concept. AUTOSAR application software components are cross-compiled for Windows environment and configured by configuration files which define their external connectivity. In addition to AUTOSAR software components, ADAPT also need Functional Mockup Units (FMU) components which represent non-software parts. In fact, FMU modules in ADAPT are based on binaries and interface descriptions according to the Functional Mockup Interface standard [16]. Then, ADAPT will trigger and provide input data into execution phase of FMU and SWC to simulate the resulting data, then collect them in order to exchange the data over the virtual signal bus again after a certain simulation time [2]. Our designs for Logger and Replayer will be implemented as integrated modules/components in this platform at the early phase of this project. A general illustration of ADAPT system is shown in Figure 2.5.

Modules or components which are used in ADAPT can be SWCs (which are primarily based on the Autosar standard), physical simulation models, logging, and even interfaces to physical buses and I/O. In order to make all execution information be inferred from models, AUTOSAR SWCs will need to use templates to define their interface and triggering properties. By using module wrappers generated from module generator in Adapt, we can use the same source code for both simulated modules and target binaries in simulation thanks to wrapper interfaces. Similarly, there is a FMU wrapper generator for creating ADAPT wrapper interfaces which will work with corresponding FMU binaries, based on its specification in the description file. The wrapper code likewise interacts with the ADAPT simulation core. The ADAPT system can also communicate with external environment using CAN and Local Interconnect Network (LIN) buses as well as IP communication channels such as UDP or TCP. These features are implemented in communication module which reads and writes from those communication interfaces. Since the communication concept is signal based, each module has an interface description file (in XML format) for declaring which signals are read and written by the module so that ADAPT can route signals correctly. Users will have to prepare these XML files prior to running the simulator so that these can be parsed correctly during simulation initialization [15].

2.7 Logging and Replaying Concept on Adapt

For the design, the function of logging as well as replaying is accomplished by developing two modules (logger and replayer) which can be run on the ADAPT simulation platform. The developed modules share the same mechanism of initialization and communication with other software or FMU running on Adapt. And the mechanism is based on ADAPT module API which can be recognized and used by both the ADAPT system side (virtual signal bus) and the module side. A brief structure explaining this mechanism is shown in Figure 2.6.

As it is shown in the Figure 2.6, the structure is divided into three parts: ADAPT system, Module and External file. The ADAPT system is referred as the VSB (virtual signal bus) along with the human interaction interface called CLI (command line interface). The CLI provides an interface with various commands that supports a lot of functions such as initialization, setting signal values and checking signal values. In Figure 2.6, when user open the CLI, the main function will be started and the commands and arguments later input by the user will be translated into function calls in the ADAPT through console executor. The called functions are located in the system executor and have various utilities including initilization and updating signal values. In the system executor these functions will be translated into general APIs of the VSB called VSB APIs. These APIs are provided by the ADAPT and each of them has a specific and detailed purpose. They provide ways to communicate with a module and to parse the information of the external XML file which describes the module.

The bridge between the VSB and the module is the ADAPT module API. It allows

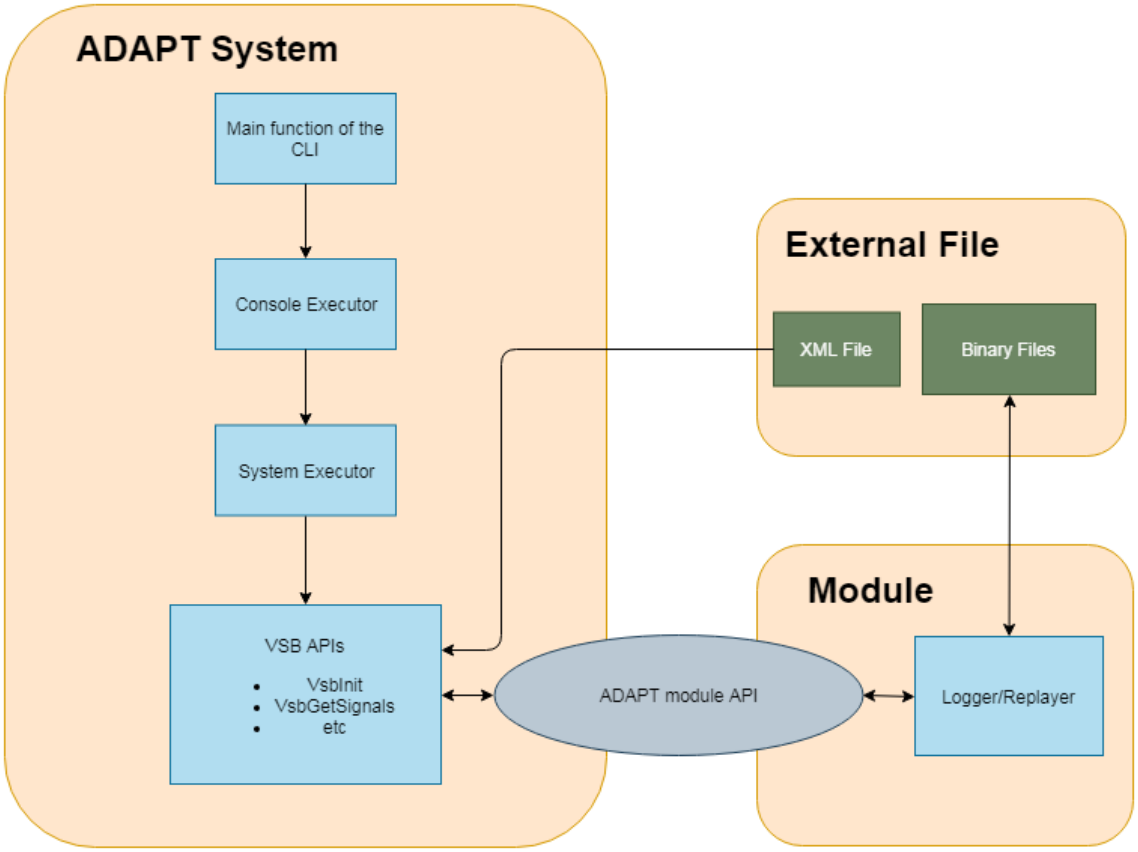


Figure 2.6: The mechanism of loading modules

the system to call functions inside the module such as initializing the module, calling step-forward function which includes output data into log files (binary files in the external file) and receiving and updating signal values from the system. It also allows the module to call functions defined in the system to send new signal values into the system.

2.8 Log File Format

The essential information we want to log from ADAPT as well as EE systems on trucks are:

- **Tick** (timestamp value): the time unit in ADAPT
- **Signal ID**: an unsigned integer number for unique signal
- **Signal Type**: data type of a signal, e.g. integer 16bits, integer 32bits, double, etc. This information can also be represented by an unsigned integer number.
- **Signal Value**: the numeric data of the signal at current Tick

For every tick value, there may be more than one signal which needs to be processed. Therefore, the file format has to satisfy this flexible condition. In general, file formats can be organized into two main categories, namely text file and binary file. A binary file can provide a complex structure and well-organized data which can help increase the speed of data processing. For example, binary file data can provide

partial access to a specific part of the file based on its memory address for reading or writing operations. However, text files are easier to handle for our components as well as more human friendly. In this thesis project, we have considered the following log file formats:

- CSV (straightforward way to store data, information is represented in ASCII);
- PCAP (binary format, open-source library available, easy to capture and filter);
- ROSBAG (format used by Robot Operating System (ROS), open-source library available);
- M4F (open-source specifications available).

2.8.1 CSV

One straightforward way to store the information mentioned above is using text file. All information will be represented as ASCII (stands for American Standard Code for Information Interchange) characters. By using this way, we can just use comma-separated values (CSV) format for our data. Therefore, delimiters, usually commas, are used to separate each field, each row reflects a record at a certain time tick and data is represented in plain text. This means that we only need to prepare and process our data in form of strings for reading or writing log data. This format is available for many platforms, and hence has an advantage compared with other formats. In addition, there are no redundant structures in CSV format since each field is fully defined by users, and external library support is sufficient.

To be practical in ADAPT environment, the possible CSV format can be implemented as shown in Table 2.1. The resulting log file will be the text file which contains ASCII character strings in m line (row). Each line consists of several fields corresponding to signals information at a Tick. The very first line of the file may contain the header which is a explanation for each field in a line. This will be useful for users who want to read the data visually in text. Then each line afterwards stores all the signals information under certain timestamp.

Table 2.1: Possible implementation using CSV format

Header in text							
Tick 1	Signal ID #1	Signal Type #1	Signal Value #1	...	Signal ID #n	Signal Type #n	Signal Value #n
...							
Tick m	Signal ID #1	Signal Type #1	Signal Value #1	...	Signal ID #n	Signal Type #n	Signal Value #n

Table 2.2: CSV format example

Tick	**Signal ID #1**	**Signal Type #1**	**Signal Value #1**	**Signal ID #2**	**Signal Type #2**	**Signal Value #2**
1	1	1	0	2	8	0.0
3	1	1	1	2	8	0.7
5	1	1	0	2	8	1.3

One example of this format for two signals can be seen in Table 2.2. While it's quite simple to implement according to description, we can optimize the format to

reduce redundant or repetitive data. Section 4.1.3 will show the further analysis and implementation result of this format.

2.8.2 PCAP

PCAP is a binary format used to capture internet packets. It provides the packet-capture and filtering engines of many open-source and commercial network tools, including protocol analyzers, traffic-generators, network-testers, etc. In this thesis, we can use this format for our data because it has libraries supporting many platforms. Actually, we can use libpcap library for software development on ECU, and use WinPcap library for Windows application [17] as in ADAPT simulator. The file structure of a PCAP file can be seen in Table 2.3 below. In this structure we

Table 2.3: PCAP file structure

Global Header	Packet Header	Packet Data	Packet Header	Packet Data	...

have two types of headers along with packet data. The global header wraps the basic information related to the whole file, including magic number for detecting the format, format version, timezone information, etc. Then there is the packet header which is exclusive for each packet. Inside it we have timestamp and length information. Timestamp information provides the regular second-precise timestamp (date and time) as well as an extra offset in microseconds. Length information includes captured data length and the original data length from the sender.

In our case, packet won't be delivered through the internet and the Logger-Replayer pair corresponding to the Sender-Receiver scenario in PCAP is designed by us. Therefore, information in global header such as format version can be pre-defined and the header can be reused. Then, the packet data may contain our log data.

In fact, our log data can be considered as a set of numbers representing Tick, Signal ID, Signal Type and Signal Value as mentioned at the beginning of Section 2.8. Therefore, the idea is that if we can arrange our data as a package of bytes in series for all the fields of log data, we can put it in PCAP packet data. By estimating the maximum range of numerical value of each field, we can definitely allocate data into the package by order of byte. As analysis, the estimations are as follows:

- **Tick**: this field is supposed to be unsigned integer, hence in range from 0 to 2^{32}. It can occupy first 4 bytes in the package
- **Signal ID**: this field has low range from 0 to 256 signals in ADAPT. So, 1 byte occupation is reasonable.
- **Signal Type**: similar to Signal ID, we will have a limit range of data type in ADAPT. So, this field consumes next 1 byte.
- **Signal Value**: the field needs to be allocated in more bytes so that we can store floating point numbers. Therefore, 8 bytes should be used.

As a result, we can prepare a package of 14 bytes containing our log data properly. Besides, we should use 16 bytes package in practice for better implementation in coding. This also add 2 bytes of reservation in case we need to modify the range, or

we can simply use 2 bytes for each Signal ID and Signal Type field. An illustration for this approach is shown in Table 2.4

Table 2.4: PCAP package

Tick	Signal ID	Signal Type	Signal Value
4 Bytes	2 Bytes	2 Bytes	8 Bytes

As mentioned above, the package has a static length of 16 bytes. This means that if we use the package as PCAP packet, the packet length is also static. In summary, our log information in PCAP will lie in a series of packets and each packet contains a set of Tick, Signal ID, Signal Type, Signal Value at a certain Tick. To demonstrate, our idea of how to make use of the PCAP format is shown in Table 2.5.

Table 2.5: Possible implementation using PCAP format

Global Header (incl. version, timezone, etc)			
Packet #1 Header (Pre-defined)			
Tick	Signal ID	Data Type	Signal Value
Next Packet...			

2.8.3 ROSBAG

ROSBAG or BAGS is a file format in ROS. This format is mainly used in ROS for storing ROS message data due to its convenience in processing, analyzing, and visualizing various kind of data such as text, image, position coordinates, etc. [18]. A ROS bag file includes an initial line indicating the current version along with a sequence of records. The record structure can be seen in Table 2.6.

Table 2.6: ROSBAG file structure

Record 1				Record 2			
Header Length	Header Data	Data Length	Data	Header Length	Header Data	Data Length	Data

Each record can contain a ROS message with different types including chunk, connection, message data, index data, etc. The very first record is called the bag header record which contains information regarding the "chunk" section and "connection" section.

All these types contribute to provision of various information and supporting ROS, however, most of them are not helpful in our case. Among them, only the type "message data" serves the purpose well since it includes a timestamp and a connection ID in the header that can be used as tick value and signal ID. This means that we can use the same header indicating the "message data" type of all records for our data. The logging information will be packaged into the message data of

consecutive records in the rosbag file. This means that all information of logging data corresponding to one timestamp (Tick) such as Signal ID, Signal Type, Signal Value are packed as message data records.

Therefore, if we use package of log data in the same way we analyzed with PCAP above, our "message data" will contain a set of data needed in ADAPT. Besides, this type of record does not require strictly defined ROS topic and ROS node, we can also use pre-defined information for these fields in headers. One example for this implementation is illustrated below in Table 2.7:

Table 2.7: Possible implementation using ROSBAG format

Header #1 Length (Pre-defined)	Header #1 Data (Pre-defined as messages)
Data #1 Length (Pre-defined)	Data #1 Data (Package data)
Next Record...	

2.8.4 M4F

M4F or MDF is a standard format supported by Standardization for Automotive Development (ASAM). This format is used to record signal data during measurement, calibration and testing from automotive application such as sensor data, ECU-internal variables/states, bus traffic in a vehicle network, or internally calculated values [19]. Therefore, the format is very promising for our research in logging data. The MDF format organize the whole file in different types of binary blocks, each one serves a specific purpose. For instance, the identification block (ID) together with header block (HD) provides an label and description for the entire MDF file. And smaller unit like signal data block (SD) or Data block (DT) provides the signal value information in different configurations.

Figure 2.7 will give an overview of MDF file structure. As we can see in the hierarchical order, the MDF file starts with ID block and its following HD block which are general information about the file such as global comments, start time of simulation, etc. The HD block will point to a Data group (DG) block where we refer to our data and its byte layouts. In fact, DG block gives us two main pieces of information: the place we put our data in (DT block) and all information that are necessary to understand and decode the data (Channel Group Block). The Channel Group (CG) block will, in turn, point to some Channel (CN) blocks so that the data in each record of DT can be read properly. For example, if we have two CN blocks (e.g. time and value), the corresponding of these two fields will be defined in each record of DT block. While MDF supports for storing data in several DT blocks, we can use just one DT block to store all records for simplicity. Therefore, a simplest MDF file may contain a set of 7 blocks which are: ID, HD, DG, CG, CN, DT.

Next, a single MDF block structure is shown in Figure 2.8. As it is demonstrated, the block is divided into three parts: header, links and data section. The block header contains basic information of the block including the type and the total size

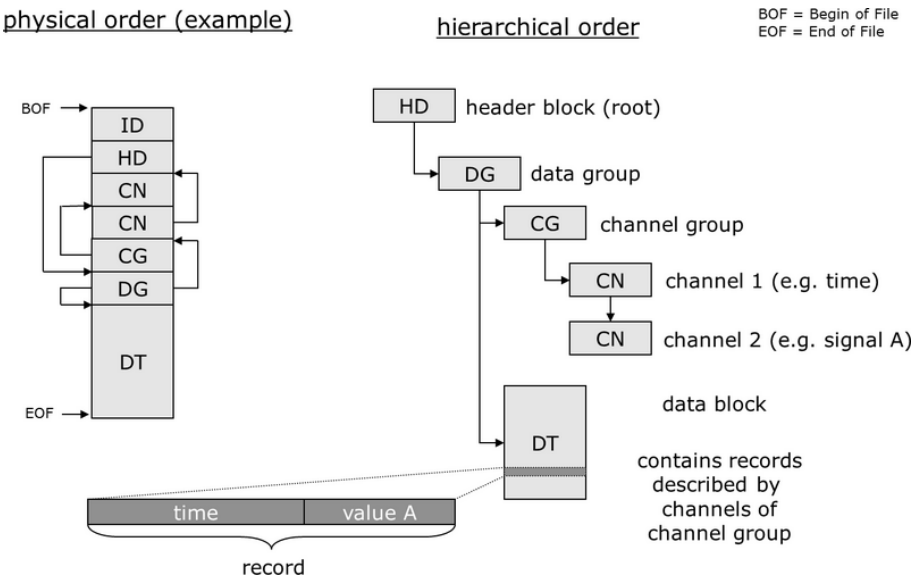


Figure 2.7: The MDF file tree [19].

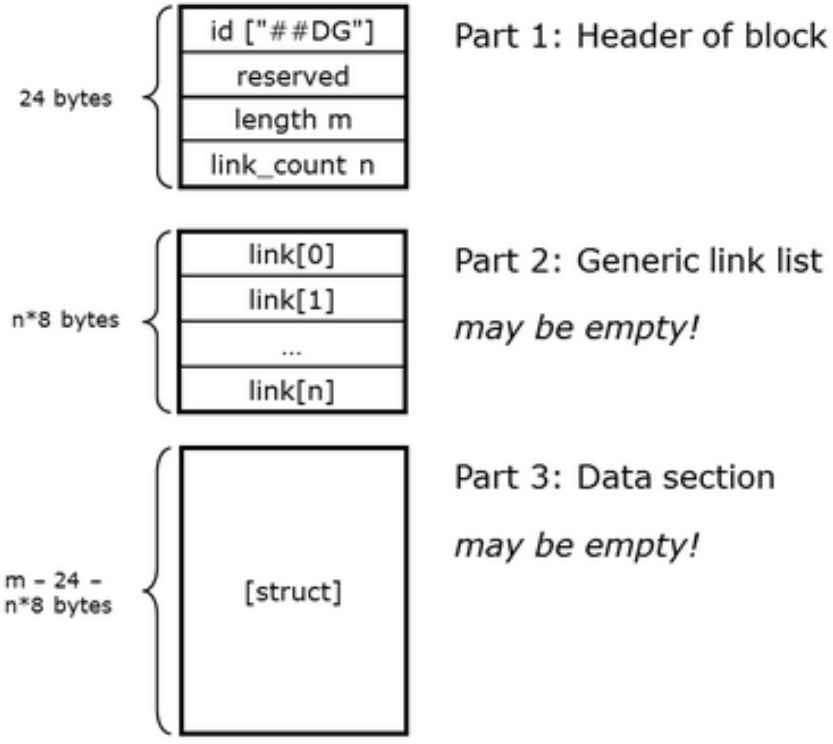


Figure 2.8: The MDF block structure [19].

of it. Then the link list is the key part that allows the MDF format to form a tree structure. Each link is linking to another block. Reflecting this structure to our logging scenario, we may use the DT block to store all of logging data from ADAPT. In this case, the DT block contains a simple header and no link. It's because our DT block is the last block of the file and will not point to any other block. The data is then a series of records consisting of our log data for each and every Tick.

One possible binary layout of DT block for three signals is shown in the Figure 2.9 below: It's worth to note that the Block Length field in the Block Header section

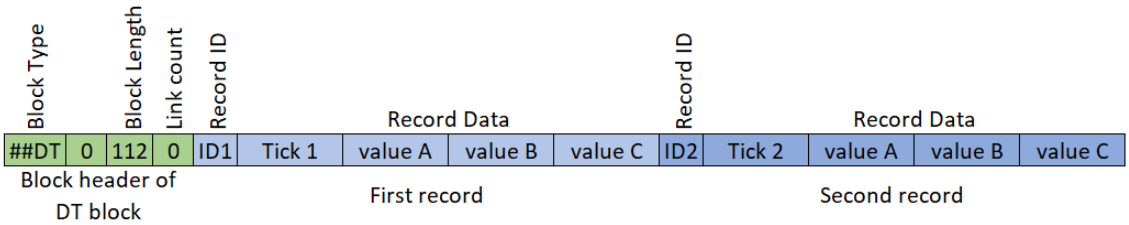


Figure 2.9: Possible binary layout of DT block

represents the total size of the DT block. So, it will be updated whenever a new record is added into the block. Besides, the value fields correspond to each Signal Value for a Tick timestamp. In this implementation, the Signal ID is implicitly defined by order of its field (e.g. value A belongs to Signal ID #1 and so on). Lastly, Signal Type can be defined in CN blocks.

2.9 AMALTHEA Platform

Amalthea [4] is a model and tool platform for automotive embedded-system engineering and provides integration into established industrial development processes. In Amalthea, a software unit is called runnable and a set of communicating runnables provides a desired functionality [20]. In general, Automotive Software projects execute the following design flow by iterating parts or the complete flow until the desired software features and quality are achieved as follows [21]:

- Collect and describe the requirements for product features, tests and dependencies.
- Define the different variants of the final product and their dependencies if needed since it is inefficient and error prone to develop each variant separately. We can use tree structure with dependencies for handling multiple variants in one project as shown in [22].
- Define the architecture of the SW system e.g., based on the AUTOSAR standard [12].
- Define the behaviour of the different software components. In this step, the functionality according to the requirements is implemented into software components. These components can be verified and simulated to prove their there correctness.
- Within the variant configuration step, the defined functionality is assembled into certain packages according to the software architecture under the variant definition.

After building a set of models for defining the final software product, we can deploy

them to an appropriate hardware platform (e.g., a microcontroller ECU). Basically, we can follow these steps:

- **Partitioning:** The software components are combined to tasks (or runnables) and these parts are put together to largely independent partitions. This step is required to be able to distribute the software to the cores of a multicore ECU [23] or even to different ECUs.
- **Mapping:** The software partitions are then mapped on the different hardware resources, e.g., a processing core of an ECU.
- **Codegen:** Program code is generated from the models by applying automatic code generators.
- **Tracing:** The overall system is verified by generating timing traces and comparing them to the desired timing of the application.

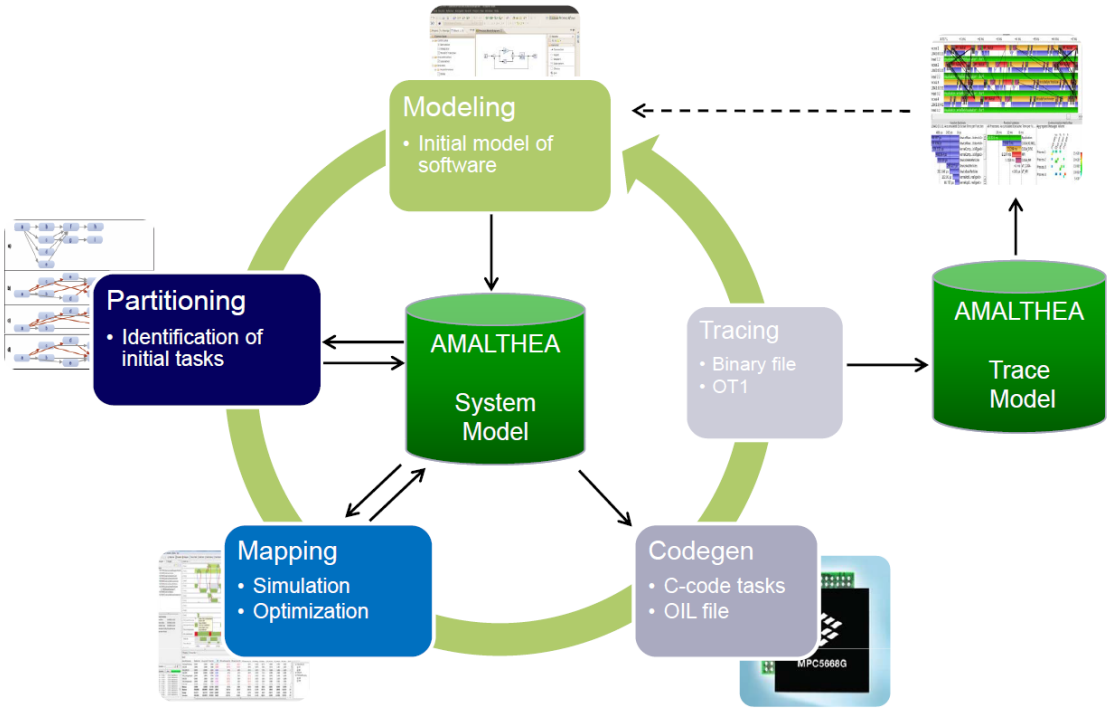


Figure 2.10: Design Flow using Amalthea platform [4].

The Amalthea platform can support all of these steps as shown in Figure 2.10. In fact, the platform is proven to be optimal in both partitioning and mapping [24]. Therefore, by using this platform for our project, we can increase the development efficiency which comes from a reducing in Development cost, Turn-Around-Time (TAT) and Error rate. As reported in [21], the TAT is estimated to decrease up to 90% for 8 iterations design works. Moreover, the error detection time within Amalthea is 95% faster compared with that of Matlab/Simulink, which also helps to reduce the error rate.

Other R&D projects have also had benefits from Amalthea in many different aspects. In [25], authors proposed new partitioning mechanism combined with both

the tracking and the tracing approach with Amalthea to optimize performance for their distributed systems. The new approach helped them to reveal errors, problems and conflicts and improve system's performance while meeting modern demands, constraints and requirements of distributed systems according to hardware and software issues such as memory accesses, cores, frequency or semaphores and timing metrics.

Alternatively, in [26], researchers used Amalthea to evaluate their proposed approach of task allocation optimization compared with well-known bin-packing approaches onto a heavy artificial task set. This set consists of 12 real-time tasks and 2,500 runnables, which executes on a processor with three symmetric cores and a clock frequency of 160MHz each. Similarly, the result in [20] also indicates that Amalthea helped to reduce significantly the number of iterations and thus the amount of manual work for reaching the design goal which targeted a heavy task set with 234 runnables and 248 communication dependencies. Furthermore, we can take the advantages of Amalthea platform from its complete continuous tool-chain and open source Eclipse project called APP4MC in comparison with commercial tools such as the TA Tool Suite [27]. The TA Tool can provide us an environment for tasks scheduling simulation and analysis. The tool also support several scheduling algorithms/systems such ash EDF, OSEK, AUTOSAR, etc.

As a result, this platform is suitable for us to develop and evaluate our designs for the Logger component in the EE system of trucks. However, we need to build the corresponding AUTOSAR software models for our design in order to adapt with this platform. The detailed implementation of Logger component based on Amalthea will be presented in the next Chapters.

3
Methods

In this chapter, we will describe the research method which we used in our project. We also explain the plan and result for each iteration of the research.

3.1 Iterative Design Science Approach

Because the development of the logging system has so many challenges as mentioned above, it is better to have a model that allow us to divide the project into iterations of task and thus we can proceed to build a complex system finally. In that sense, we will apply iterative design science approach based on [28] for the research of logging technology on trucks. The approach consists of three main cycles which are Relevance, Design and Rigor cycle.

We may start with Relevance cycle where we identify the requirements of the logging system and define the evaluation criteria of all research results. After having concrete requirements and criteria, we will perform literature reviews for current logging technologies in Rigor Cycle in order to grab the essential knowledge including existing methods, processes and theories which can be used as a knowledge base in our research. For example, the knowledge for ADAPT, WCET tools and Amalthea will be studied in this cycle in many different iterations. We will also update our solution after each iteration based on the outcome of our evaluations, experiments, discussions and results from iteration literature studies or artifacts.

In Design Cycle, we can use this base knowledge to develop the product designs for the logging system in order to satisfy the requirements from Relevance Cycle. For each iteration, we define a set of accepting points or local criteria which is a subset of the acceptance criteria in Relevance Cycle corresponding to the current research iteration. Then, we use these local criteria to evaluate the current design or method. The part of studying and selecting the WCET tools is a typical example of this process, where we have to perform some local evaluations from many existing WCET tools in order to find out the most suitable one for this project. If we cannot meet the local evaluation in an iteration of the research, we need to improve the knowledge base back in Rigor Cycle to update our artifact design. The research step will move to Relevance again once the design or method satisfy evaluation criteria in Design Cycle, and will be considered further depending on overall status of the research.

Through this developing model, our research project can proceed in a more robust way by iterations. Our first few iterations will cover the encoding and annotation of the log data as well as the basic logging and replay functions. After the basic functions are satisfied, we will start with more advanced functions including implementing data label, pre-processing in real-time, filtering, etc in the later iterations. In each iteration, we update both our requirements and knowledge through Relevance cycle and Rigor cycle. Figure 3.1 below shows the summary of our approach.

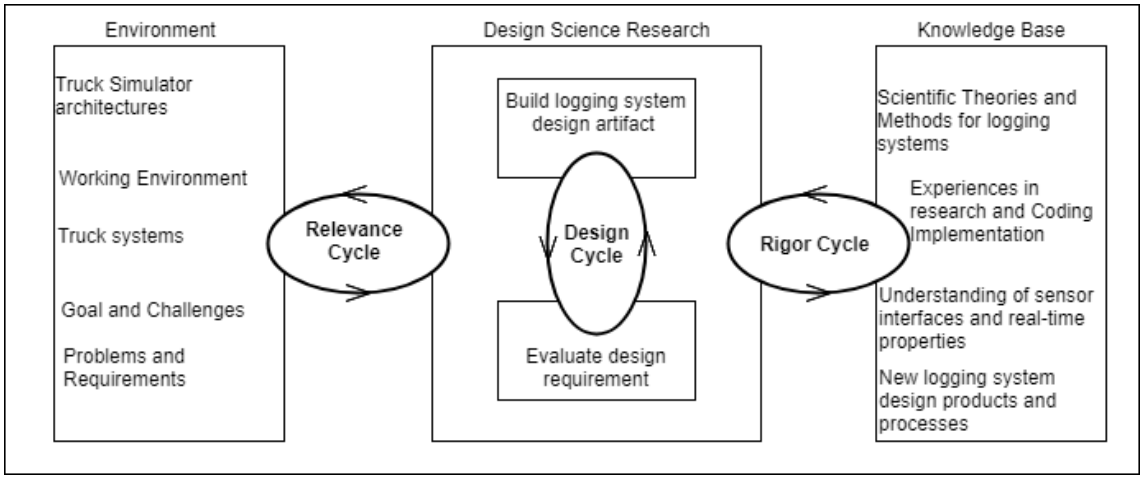


Figure 3.1: Design Science Research Cycles adopted from [28]

3.2 Design Choices and Iterations

Starting with software designs for logger and replayer components, we followed several iterations which are described in the sub-sections below.

3.2.1 Iteration 1

In the first iteration, we investigated the ADAPT system environment. The main purpose is to analyze and examine the Application Programming Interface (APIs) provided by ADAPT system itself. For this stage, we had not considered the real-time property of the design yet because the simulator is a Windows platform's software which does not require any real-time constraints. As a result, we only need to investigate the algorithm, structure of the components and input/output (I/O) interaction of our design.

Result: After this iteration, we finished the software component design which can be integrated into the current ADAPT system. The detailed design and analysis are shown in Section 4.2. The I/O information in this stage are only standard inputs and outputs from a terminal, which can only proceed manually.

3.2.2 Iteration 2

After building the first version of software components and checking the I/O results, we need to define an efficient file format so that the components can read and write during their operations. There are some candidates for the format which are CSV, PCAP, ROSBAG, M4F. We need to investigate their properties as well as advantages/disadvantages for each candidate.

Result: The result is shown in Section 2.8 and Section 4.1. We also tried to implement these formats into the logger and replayer components. However, we can only succeed with CSV, PCAP and our custom format which we proposed after trying implementations these candidates.

3.2.3 Iteration 3

The next step is to establish the virtual EE environment corresponding to the real trucks system so that we can deploy and evaluate our design in a real-time environment. We decided to use AMALTHEA and TA Tool Suite for this establishment because these tools are available and/or easy to get support from Volvo. The supporting toolchain of AMALTHEA and TA Tool Suite is recommended because we can acquire license from manufacturers in time.

Result: We build up the environment which can build, simulate and evaluate our design in real-time aspects which is one of our original research question.

3.2.4 Iteration 4

We built up the task set for our system to evaluate the design. At this point, we need to consider the tools for WCET problem. We tried with some different tools aiming to WCET estimation namely aiT tool, RapiTime, SWEET, ect. We got some problems with acquiring licenses for the tools and we could only wait for responses from the corresponding supplier companies. While other tools seems to be promising in solving WCET in general case, we decided to use aiT tool for static analysis of WCET due to their dedicated supports. We also tried and come to design choice of a custom format at this iteration. The format helps to improve the quality of our design.

Result: We estimated WCET of logger and replayer in a simple design with static analysis. From these information, we built up successfully the task set and hence finish simulation and evaluation properly.

4
Results

This chapter will present our research and implementation results. It starts with our design choice of log file format in Section 4.1 followed by Section 4.2 where we explain our design for ADAPT system. Next, we establish a EE subsystem and describe our components design within Section 4.4. Last but not least, Section 4.5 will show the evaluations for our designs.

4.1 Log file format and structure selection

4.1.1 Formats analysis

After considering the logging file formats mentioned in Section 2.8, namely CSV, PCAP, ROSBAG and MDF, we found that some of them (i.e. PCAP, ROSBAG or MDF) are very promising in logging and replaying ADAPT data thanks to their dedicated functionalities. Their excellent effectiveness was recognized in many other research such as [17] or [18]. However, they also have many drawbacks which prevented us to pursue these solutions for log file format.

One problem is that some of them required a complicated design to adapt with the current software component design in this project. Taking the case of ROSBAG for an example, it requires a similar setup to ROS to work with. This means that the ROS messages will be broadcast by a publisher (sender) to subscribers (receiver) for each topic (message type). This is quite irrelevant for our system which only collects data from a subsystem.

For MDF case, the problem lied in its complexity and difficulties in implementation. While the format specifications are an open-source documentations, the shared external libraries for implementation are limited. This caused many difficulties in adapting this format into our designs because we mostly build up the design library from scratch.

In addition, PCAP, ROSBAG and MDF contain many redundant features which are not our regions of interest such as Internet Protocol (IP) addresses in PCAP, ROS topics in ROSBAG or generic link list in MDF. Many features consume redundant data which would simply be wasted if applied to our project. Besides, it is the fact that these formats need to use external libraries may make our design become too complex for further analysis such as WCET estimation as in Section 4.3.

CSV format, in the other hand, is very easy to implement thanks to standard libraries. However, this format is generally less compact and lacked functionality. The format also depends on how we construct our data in the file such as what should be stored in each field, how many character for a data value, etc. If we want to use this format, we should consider to optimize the data structure beforehand.

4.1.2 Format File Comparison

From analysis above, we can summarize the research on logging format as shown in the Table 4.1.

Table 4.1: Logging formats comparison

Logging Format	Implementation	Size Compact	Redundant Feature	Library support
CSV	Simple implementation. Information can be represented as ASCII with different field separated by commas, so that the data can be prepared as string.	Unlikely to be small as ASCII representation is used.	No redundant feature as all the elements in the format are either recording the information or used for categorization.	YES
PCAP	Generally easy. Open-source libraries are available and since they are written in C, no extra effort is need for implementation. in a C++ environment.	Smaller size than CSV as data is represented in binary.	As originally designed for network data, unnecessary overhead such as data link type and timezone information are also included.	YES

Table 4.1 continued from previous page

Logging Format	Implementation	Size Compact	Redundant Feature	Library support
ROSBAG	Similar to PCAP, open-source library is available and can be used without too much effort.	Smaller size compared to CSV due to binary representation.	ROSBAG provides a lot of features such as reindexing and compress/decompress, these can be generally useful and may not be used in our project due to unnecessarily complex.	NOT for Windows
MDF	Complex file structure with more than 20 different data types makes it hard to implement in our project.	Small memory consumption due to binary storage.	Many data types are not necessary to use in our case.	NOT for linux

Therefore, from these analyses, we come to the design choice where CSV is one of our final solution for logging file because it can be very easy to handle and implement.

4.1.3 CSV and custom format data structure

As mentioned above, we have decided to use CSV as our log data format. However, CSV consumes a lot of disk space to store data. This leads us to consider a more efficient structure for our data.

In fact, our signal data can be stored in fields including only Time (tick) and Signal Value. The idea is that we can define implicitly the information of Signal ID by organizing data as columns. This means that the value in first column corresponds to the signal whose ID is 1, and so forth. Besides, the Signal Type can be determined by the floating point format and the numeric magnitude of the value itself. For example, an uint8 (unsigned integer 8 bit) value will be in range from 0 to 255, and no floating point. Besides, when the logger/replayer initialize, they will read input configurations from xml files in order to unify the list of signals and the order among them, and hence it's reasonable for us to implement this format.

Therefore, we can create a CSV structure that is simple enough to reduce the data

size. The structure can be seen in Table 4.2. The separators (SEP) will be used to separate each field in CSV format. The preferable SEP, comma (,) character, will be used in this implementation.

Table 4.2: CSV Data Structure

Tick 1	SEP	**S1 value**	SEP	**S2 value**	SEP	**S3 value**	SEP	**S4 value**	**...**
Tick 2	SEP	**S1 value**	SEP	**S2 value**	SEP	**S3 value**	SEP	**S4 value**	**...**
...									
Tick n	SEP	**S1 value**	SEP	**S2 value**	SEP	**S3 value**	SEP	**S4 value**	**...**

Although we established a reasonable data structure for CSV, the log file is still not optimal in size. In fact, the numeric values of Signal Value will be stored as ASCII strings. This means that we need to use many ASCII characters for rational numbers corresponding to Signal Values. For example, the numeric value of "1.2" and "1.2345" requires respectively three and six characters to represent in CSV. Therefore, the more significant figures a numeric value has, the more characters we need.

This problem makes us consider the second feasible format which is derived from the original CSV data structure. If we can organize the data of Tick, Signal ID, Signal Type and Signal Value in a fixed number of byte data, the information can be packaged as binary file where data will be stored serially. By this way, we will not need any separator as in CSV format and help reduce the file size. As a result, our second solution can also satisfy both simplicity and size requirements.The format is shown in Table 4.3.

Table 4.3: Custom format

Tick	Signal ID	Signal Type	Signal Value
4 Bytes	4 Bytes	4 Bytes	8 Bytes

At this point, we would like to take a note that we store information explicitly (Signal ID, Signal Type, Signal Value - Table 4.3) instead of implicitly (only Signal Value - Table 4.2). The main reason is that when we store information in series of byte, it's very difficult to distinguish which byte represents Tick or Signal Value when we read the log file. This comes from the fact that the total number of signal is not a constant but a variable from input configuration. Although we use more data in the log file for the custom format (20 bytes instead of 8 bytes for each signal), it will help increase performance. This is because we can divide log file into chunks of 20 bytes so that they become easier to process. Actually, the implementation is even optimized more when we use standard data structure such as vector to implement the format. Therefore, we will use this structure for further analysis.

4.2 Implementation of logger and replayer on Adapt

4.2.1 Introduction

As we have mentioned before, the logger and replayer are working as modules inside the ADAPT simulation platform. To work as a module, a configuration XML file including the log file information and the signal information will be provided along with the software. Also, modules inside the simulator provide callback functions by exposing the external APIs through a Dynamic-Link Library (DLL) file. Among the external APIs, there is one for initialization in which the configuration file is utilized to start the execution of the logger and replayer. The configuration file is parsed in the very beginning of the initialization callback function and both modules perform the same operation in this part.

After the parsing of the configuration file, the signal information including Signal Name, Signal ID and Signal Type and the log file information including file name, the start and end position of data and the file format are extracted. Among all the information, the one belong to the signal is the most important since it has to be read every time when the logger is updating the data for a signal or the replayer is reading data from the log file. In order to save the signal information as well as achieving better performance when the information needs to be read or updated, we choose to use the associative container std::map in our case. Associative containers can be more efficient than sequence containers in accessing value by the key rather than by value [29] and it suits our case well since the logger and the replayer are unaware of in which order the information for each signal is stored. By practice, in the initialization, we do mapping from the signal id to its name, value and declaration for both modules. After mapping the signal characteristics, the initialization process now differs between the logger and the replayer. In the logger, two more pre-logging actions have to be done. First, the logger generates a compare string which is later being used for judging if certain signal data needs to be updated or not. Then the log file information mentioned above is parsed and stored in the RAM for later usage. In the replayer, the pre-stored log data is fed by referring to the file name.

When the init function is finished, the logger and replayer are now operating by a certain time interval. Regarding the time interval, the simulation environment maintains the updating of a tick value and the updating rate of the tick value determines the fastest rate both the logger and replayer can operate. In our case, the fastest rate is one operation per millisecond. To implement this, a step function is triggered whenever the tick value is updated. And within the function, the logger writes the information saved in string to the log file and the replayer writes the data to the simulation environment. However, not all the signals have to be updated at the highest frequency as it would cost too much computation power as well as generating redundant data for the log file. To cope with this, a pre-condition, which only activate the update at desired tick value, is implemented for each signal.

Except for the init and step external API that we have mentioned before, there are other APIs for writing signal data to module, writing signal data to system, getting signal id, etc. The most important that we want to mention here is the one intends to write signal data to module. It is a callback function which is implemented on the module side and is for writing signal data from the system to the module. In the argument list, the simulation environment would supply the signal id and the signal value to the function. For the logger, these two parameters are taken to update the output string. However, since replayer is barely mimicking the operation of a module, this callback function is simply a dummy one.

4.2.2 Flow Chart

For better illustrating our implementation for the logger and replayer, we have the basic mechanism described in Figure 4.1 and Figure 4.2.

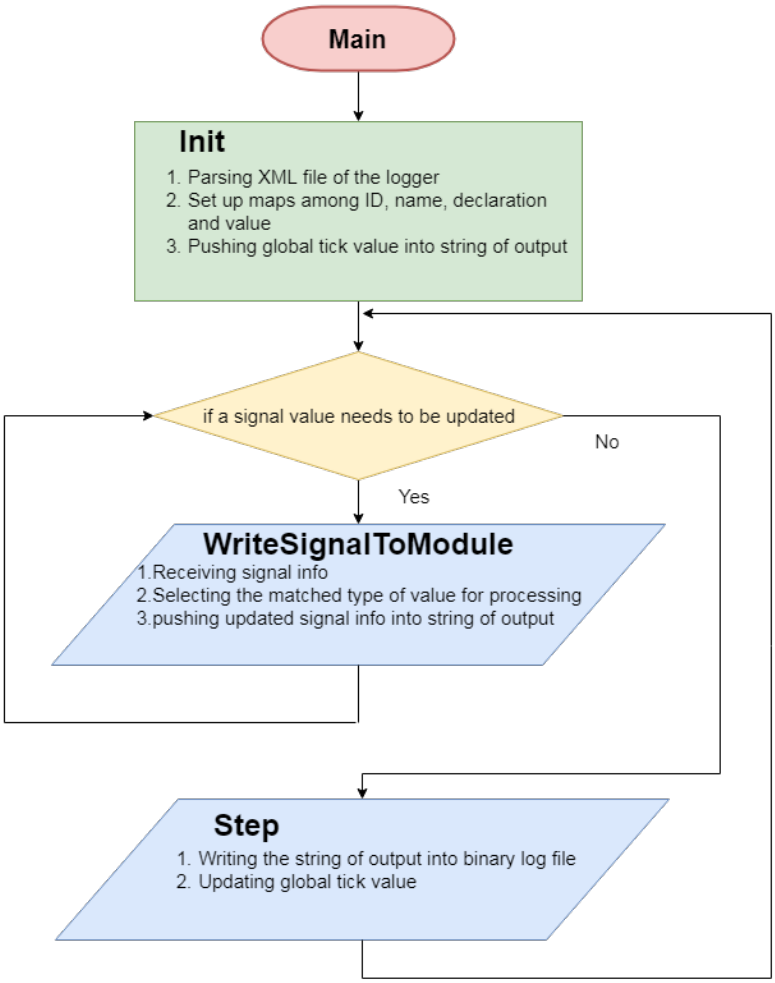


Figure 4.1: The mechanism of the logger

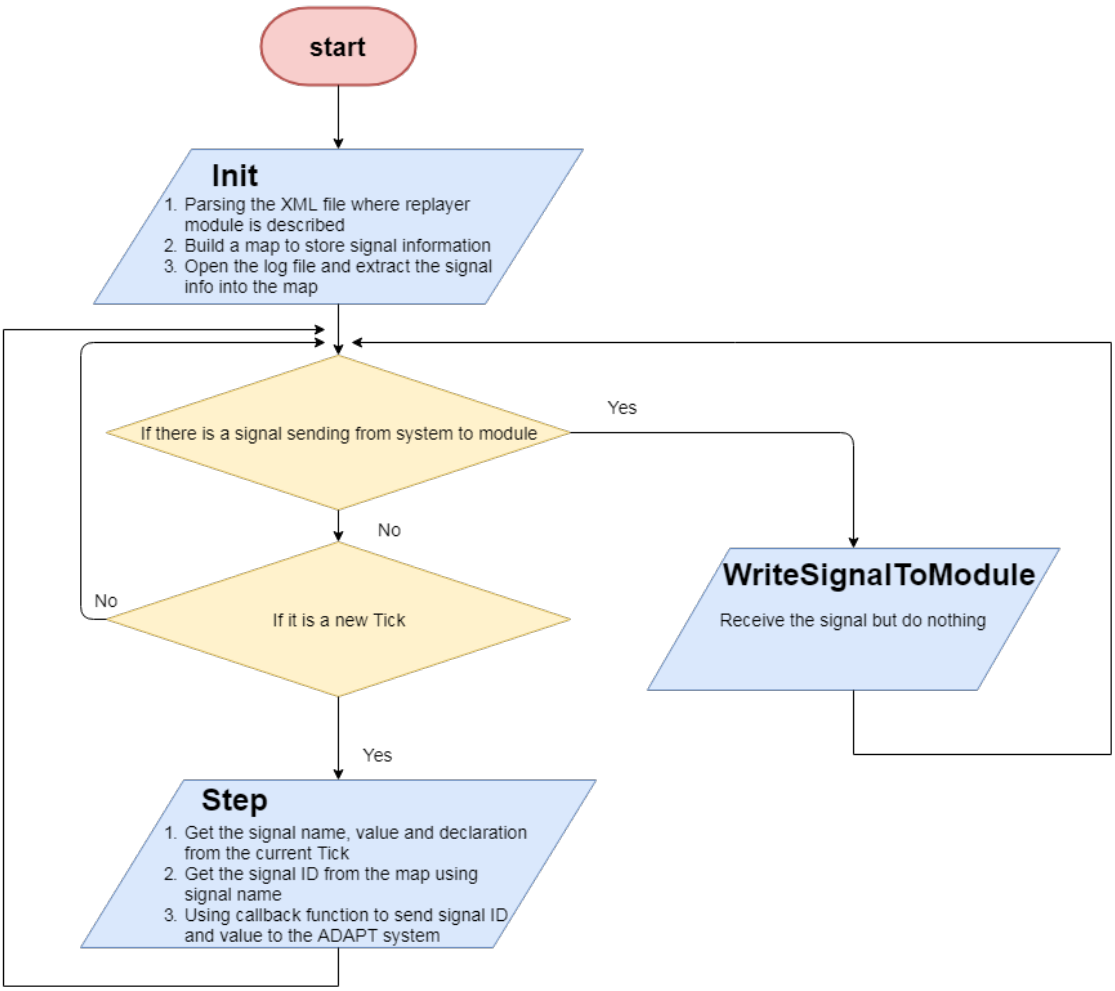


Figure 4.2: The mechanism of the replayer

As it is shown in Figure 4.1, when the logger is started, it firstly enters the Init phase. In the Init phase, an XML file describing the signal receiving and sending in this module will be loaded and parsed. Also mappings are set up to store signal information including name, ID, declaration and value. After the Init phase, the logger will enter a loop to write out data into binary log files for every tick (time unit in the ADAPT environment). And in each tick, updated signal values from the system are received and stored in the output buffer.

In the replayer, there is also a Init phase to parse the XML file, set up the mappings of signal information and extract the information from the log file. And there are two loops in the replayer. One is a dummy function to receive the new signal values from the system but do not perform any processing. Another is to sending out signal values read from the log file to the system for every tick.

4.3 Components WCET Estimation

After we finish logger/replayer designs in Section 4.2, we can reuse these designs in EE systems on trucks. The problem is that when we design new software components in such systems, we need to verify their realtime properties where our designs must meet both functionality and hard deadline requirements. This required us to perform WCET for our components in order to evaluate in Section 4.4. As we have mentioned in Chapter 2, the aiT tool from AbsInt is been used to perform a WCET estimation for our logging implementation that is intended to be deployed on the EE systems of trucks.

The aiT tool considers facts including memory accesses upon cache misses, pipeline states of the processor, etc. However, since the external storage, e.g., hard drive, varies among different EE systems, file stream I/O operations cannot be covered by aiT. As a result, we perform the estimation for the logging process without writing operation towards external storage i.e., the WCET estimation of the internal process of the logging component. For the external storage I/O cost, we consider the practice of an AUTOSAR platform ECU with CAN bus since it is commonly used in automotive industry. Calculation is made based on existing results.

For the internal process of the logger, it includes initialization, sorting out messages, etc. In order to analyze our logger implementation exclusively, we have to exclude the ADAPT environment based on which the component is operating. It creates a problem since our logger have to set up connection and initial configuration along with the ADAPT environment to be able to work properly. To resolve this problem, the dependency of the ADAPT environment has to be removed. As it is described in the background, our logging software works as a module of the ADAPT system and the connection between them is the ADAPT module APIs. So instead of calling APIs from the ADAPT system, alternative functions are implemented and can be called directly on the logging module (the one used for estimation). Besides, since external file operation is excluded in the WCET estimation by aiT tool, the parsing

process which extracts configurations including signal information from XML files also needs to be dealt with. Alternatively, we hard coded the configuration from the windshield wiper module which is used within Volvo truck. The result from aiT for a single signal (using the custom format with 20 bytes per signal) is shown in Figure 4.3 as below.

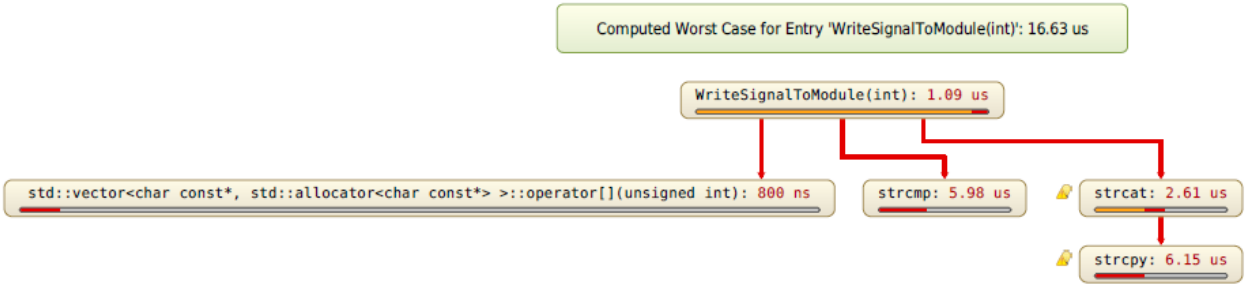


Figure 4.3: WCET of the internal process of logger

As we can see in the figure, in the worst case scenario, it takes 16.63 micro seconds to log the data for a single signal and prepare for the information ready to be stored.

After we acquired the WCET for the internal logging process, we estimated the external storage I/O timing cost. We refer to a existing implementation [30] of the AUTOSAR platform with CAN bus for data transfer. As we have mentioned in the background, the AUTOSAR platform is composed of software components(SWC), runtime environment(RTE) and basic softwares(BSW). And our logger, as an application for the ECU, will be a SWC in the platform running on the RTE. As it is shown in Figure 4.4 below, in order to write data to the CAN bus, data from the SWC has to firstly go through RTE and then the communication module which is one of the BSWs. Finally, the data is written to the CAN bus.

In this implementation that we referred to, the message that is used to transmit through CAN communication is a 32-bit little-endian CAN message. After summing up the delay on every stage, it takes in total 36.05 micro seconds to transmit 32 bits of information and thus 9 micro seconds for 1 byte. Therefore, since we are using the custom log format of 20 bytes, the time cost on the external storage can be calculated with Equation 4.1 below.

$$20\ bytes \times 9\mu s = 180\mu s \tag{4.1}$$

Now, considering the time spent on internal process as well as writing out through CAN bus, for logging a single signal, the WCET estimation of our logger is 196.63 micro seconds.

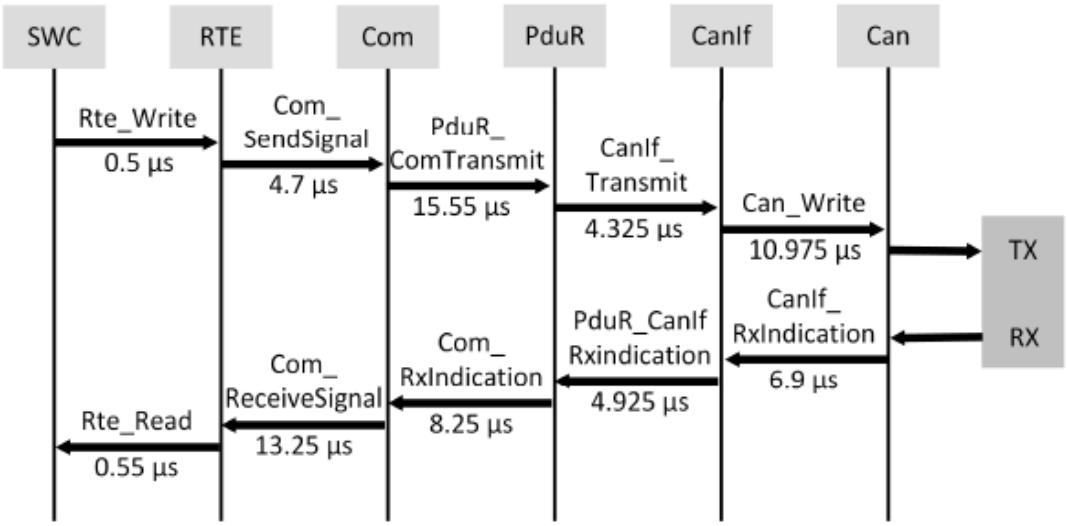


Figure 4.4: External storage time cost [30]

4.4 Component Design for hardware simulation

At this point, we would take a note that we will not evaluate our components directly on EE system on trucks i.e. real hardware ECUs. This is because the target building and running on ECUs require more time to implement and evaluation since our software components will not run solely but in a harmony with all other software components. In fact, any new software component introducing to EE system on trucks need testing through many phases. This, however, is out of scope in this project. Therefore, we establish a virtual subsystem which is a EE system yet smaller scale as shown in Section 4.4.1. Once we build up such subsystem, we can simulate for schedulability of our designs in Section 4.5.2.

4.4.1 Virtual subsystem of EE system on trucks

We established a system which can evaluate our design in real-time perspective. The system will take into account several aspects from hardware, real-time operating system/scheduler, runables/tasks set, etc. By using supporting features from AMALTHEA and TA Tool Suite, we can create an environment in which our design will be examined and evaluated.

4.4.1.1 Hardware Model

The hardware model which we are using in this project is based on Infineon Aurix TC297T [31] processors. The design choice comes from the fact that these processors are available and currently in use as platforms for deploying software at Volvo. The processor includes three cores ECUs operating at 300MHz frequency. The model can be shown in Figure 4.5.

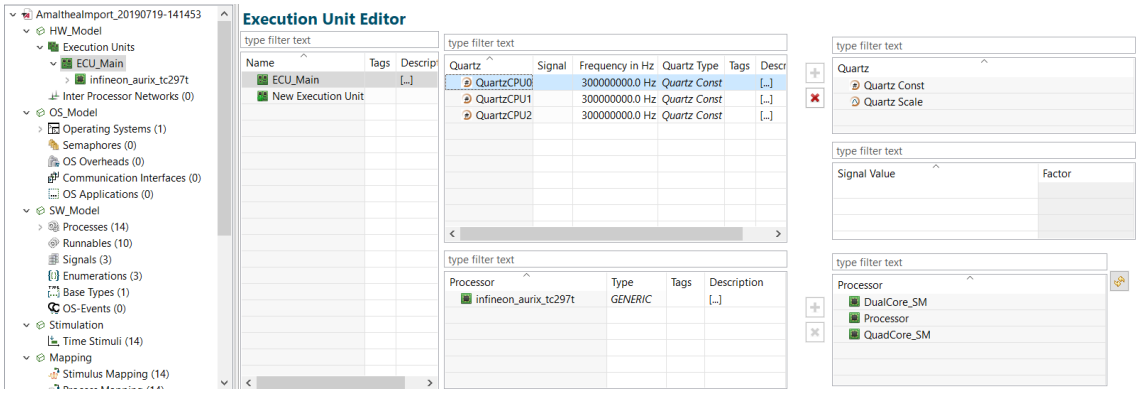


Figure 4.5: The Hardware model for ECU on trucks

4.4.1.2 Software Model

In the software model, we use a generic operating system (OS) which using EDF algorithm as task scheduler. The design choice for EDF algorithm for this stage comes from the fact that its guarantee bound is 1, i.e. 100% [7], which is relative higher than that of RM or DM algorithms.

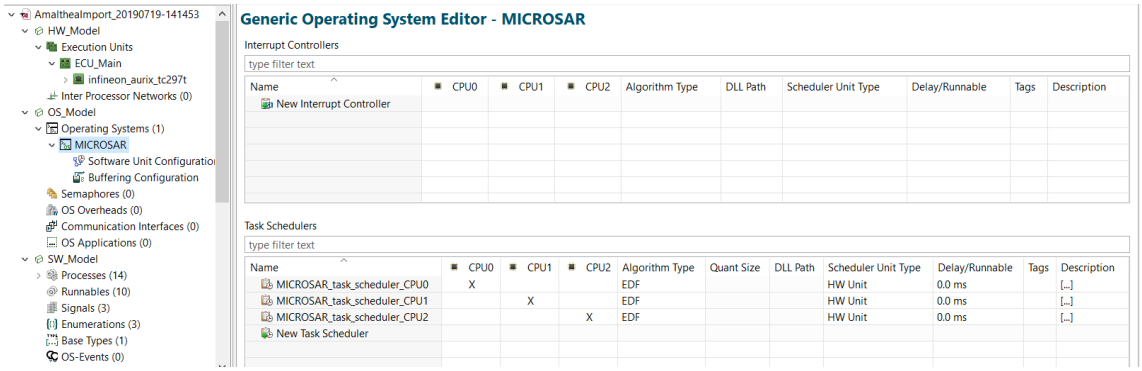


Figure 4.6: The software OS model

After defining OS model, we continue with building the software component model. This includes the processes/tasks set for the whole system. Obviously, the logger and replayer components will run along with another software components so that they will collect and send data with these components. However, the EE systems on trucks contain a huge amount of software components, which we could not handle all of them by once.

To perform analysis and evaluation our design, we have to put our design under some practical situations because the logger and replayer will never work separately with others. With support from Volvo, we decided to use three software components which are Brake-by-wire (BBW), Windshield Wiper, and logger/replayer. In this setup, BBW has higher priority than that of Wiper, logger and replayer. The logger and replayer will operate alongside with the Wiper and aim to send/collect signals data from this component. This means that the Wiper will receive signals from

replayer, perform the corresponding actions while the logger will record the signals accordingly.

Therefore, we need to analyze the tasks set corresponding to the software components above. From the task set, we can perform real-time analysis of the subsystem so that we determine whether our software components satisfy a hard real-time system or not. While the processes/tasks' properties of BBW and Wiper components can be obtained from the existing designs of Volvo, those of logger and replayer components need to be analyzed from our design. This means that we have to determine the values of Period(T_i), Deadline(D_i) and WCET(C_i) of our components. For convenience, in this project, we arbitrarily set the Windshield Wiper (or XFunction for convenience), logger and replayer to work at a resolution of 1ms which is faster than normally expected frequency. Also, it's reasonable to choose $T_i = D_i = 1$ms for the above three processes since period can be the same with their deadline considering their usage. The remaining values are WCET properties which were estimated using AbsInt aiT tool.

At this point, it's worth to take a note that the WCET estimation values here come from logger/replayer design using CSV format rather than custom format as shown in Section 4.3. This is because when we conducted the research through Section 3.2.4, we realized that we could improve design performance with custom format. In fact, the WCET estimation value of custom format is less than that of CSV format. This means, if our designs can be scheduled successfully using CSV, we can do the same for custom format. The reason is that, our designs will cost less time to finish execution in case of using custom format.

As a result, we built up a tasks set as shown in Table 4.4. The table also introduces the Utilization value, $U_i = \frac{C_i}{T_i}$, which is one of the important measurements for further analysis.

4.4.2 Tasks allocation and scheduling

The tasks set from Section 4.4.1.2 will be assigned into different processors. In this stage, we reused the tasks allocation for BBW component as in Volvo's designs. The tasks from τ_1 to τ_{11} are pre-assigned into three processors (P_0, P_1, P_2) of the Infineon Aurix TC297T ECU hardware model. The BBW tasks allocation is shown below:

$$P_0 = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_8, \tau_9, \tau_{10}, \tau_{11}\}, U_{P0} = 0.1 \times 8 = 0.8,$$

$$P_1 = \{\tau_7\}, U_{P1} = 0.1,$$

$$P_2 = \{\tau_5, \tau_6, \}, U_{P2} = 0.1 + 0.1 = 0.2$$

Next, we need to assign remaining processes, namely XFunction, logger and replayer, into the three processors. We chose the EDF-FF [32],[33] algorithm for this assignment because of several reasons. Firstly, we cannot assign any other task into P_0 because the current utilization of this processor is 0.8. If we assign any other task among wiper, replayer or logger, its utilization value will be larger than 1, which

Table 4.4: The Software Model tasks set

Task	Task ID	T_i (ms)	D_i (ms)	C_i (ms)	U_i (%)
ABS_FL_Pt	τ_1	5	5	0.5	0.1
ABS_FR_Pt	τ_2	5	5	0.5	0.1
ABS_RL_Pt	τ_3	5	5	0.5	0.1
ABS_RR_Pt	τ_4	5	5	0.5	0.1
pBrakePedalLDM	τ_5	2	2	0.2	0.1
pBrakeTorqueMap	τ_6	3	3	0.3	0.1
pGlobalBrakeController	τ_7	4	4	0.4	0.1
pLDM_Brake_FL	τ_8	6	6	0.6	0.1
pLDM_Brake_FR	τ_9	6	6	0.6	0.1
pLDM_Brake_RL	τ_{10}	6	6	0.6	0.1
pLDM_Brake_RR	τ_{11}	6	6	0.6	0.1
pWipingFunction	τ_{12}	1	1	0.3	0.3
pLogger	τ_{13}	1	1	0.36	0.33
pReplayer	τ_{14}	1	1	0.35	0.3

Note 1: ABS="Antilock Brake System", p/pt="prototype", LDM="Local Device Manager", FL="Front-Left Wheel", RR="Rear-Right Wheel", and so on.
Note 2: In fact, 1 ms rate of logger/replayer is typically too fast. While this does not change the methodology, it should be noted that the figures are unrealistic, and hence, it's used for demonstrating the system capacity purpose.

will cause failure for the system. This means that we can only allocate the three processes into another two processors. Secondly, each of P_1 and P_2 was assigned 1 and 2 tasks respectively, thus we cannot assign all three tasks in to P_1 or P_2 due to excess 1 utilization problem. This means that we can only assign three tasks in $1:2$ ratio into the two processors P_1 and P_2. Aiming to minimize the resulting total utilization of each processor, the suitable allocation can be seen below.

$$P_0 = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_8, \tau_9, \tau_{10}, \tau_{11}\}, U_{P0} = 0.1 \times 8 = 0.8,$$

$$P_1 = \{\tau_7, \tau_{13}, \tau_{14}\}, U_{P1} = 0.1 + 0.36 + 0.35 = 0.81,$$

$$P_2 = \{\tau_5, \tau_6, \tau_{12}\}, U_{P2} = 0.1 + 0.1 + 0.3 = 0.5$$

Finally, we can apply the Liu and Layland for RM [6] schedulability test with the upper bound given by

$$U_{RM}(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1),$$

where n is the number of task, for the tasks set. We found that:

$$U_{P0} = 0.8 > U_{RM}(6) = 0.735$$

$$U_{P1} = 0.81 > U_{RM}(3) = 0.78$$

$$U_{P2} = 0.5 < U_{RM}(3) = 0.78$$

Therefore, the Liu and Layland tests for RM have failed for P_0 and P_1, and hence, we should use other algorithm rather than RM or DM. This leads us to use EDF [7] because

$$U_{EDF} \leq 1,$$

which all three utilization values of the processors can meet. Furthermore, the tests with EDF upper bound are both sufficient and necessary. As a result, our task set and allocation are schedulable by the EDF algorithm. In other words, our design can meet the hard-realtime constraints within this subsystem. Notice that this analysis here provides a theoretical proof, the actual scheduling policy might not be chosen freely in real scenario and processors may also need more redundancy with lower loads.

Again, it's worth to note that CPU loads as high as 0.8 are not desired regardless of policy in practice. It's because the higher CPU loads, the more risk of failure a system will fail due to less margin load to handle unexpected events such as memory failures or network failures. Therefore, the methodology as shown above aims to give a proof of concept that our implementation is feasible, and that our design can work in theory. The next section will provide an overview on simulations and practical approaches.

4.5 Design Evaluations

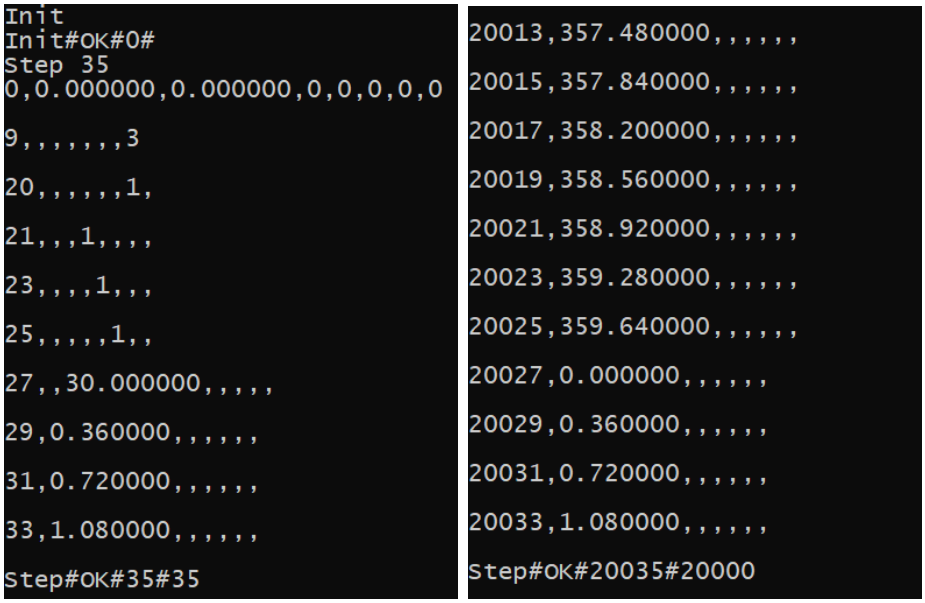
4.5.1 Logger and Replayer on ADAPT

The evaluation phase for Logger and Replayer components on ADAPT is quite straightforward. We can check manually the results from logged file compared with predefined expected data. Alternatively, we can evaluate the usage of several format between the logger and replayer so that the components can read or write the same data even when they are working with different formats. For example, we can read logged data from CSV file in Logger and write to file in custom format in Replayer. The results are shown in Figure 4.7.

The console outputs illustrate the way in which our data are constructed in CSV. There are series of lines containing numbers and comma characters. The first number is Tick value followed by signal values. And since only the changed signal value will be logged, there are empty values (no number shown between two commas). We also tried to test for the case that we run ADAPT in a long simulation for more than 20000 ticks. As a result, our design satisfied all requirements for ADAPT and the functionality is evaluated properly.

4.5.2 Schedulability Simulation Results

As discussion in the beginning of Section 4.4, we need a virtual subsystem so that we can evaluate our designs. After establishing such system in Section 4.4, we can



(a) (b)

Figure 4.7: (a) Console output from ADAPT (b) Console output continued

exploit the ability of TA Tool to run the schedulability simulation. One of the advantages of the tool is that we can try many variants of subsystem and check the result quickly without manual heavy calculations for schedulability. The tool also offers some scheduling algorithms/OS such as OSEK [34], pfair [35], AUTOSAR, etc. In fact the simulation results confirmed our analysis in Section 4.4.2. As we can see in Figure 4.8 and Figure 4.9, the tasks set meet all requirements and no violation occurs.

In details, the Gantt chart shows when a task starts and waits for processor (in gray area), or is being preempted by other tasks (light green area), or executes and finishes (dark green). The chart also gives us the information for other instances of a task after its corresponding period.

While all task requirements are fulfilled as shown in Figure 4.9, there is a notation in the metric column of the result. Here, we used the term "Maximum Response Time" instead of deadline metric. However, the two terms are equivalent because the upper limit of response time is the deadline itself. This comes from the fact that response time is defined as the time from a task starts until its completion. Therefore, if we set the upper limit of response time as deadline value, they will become equivalent.

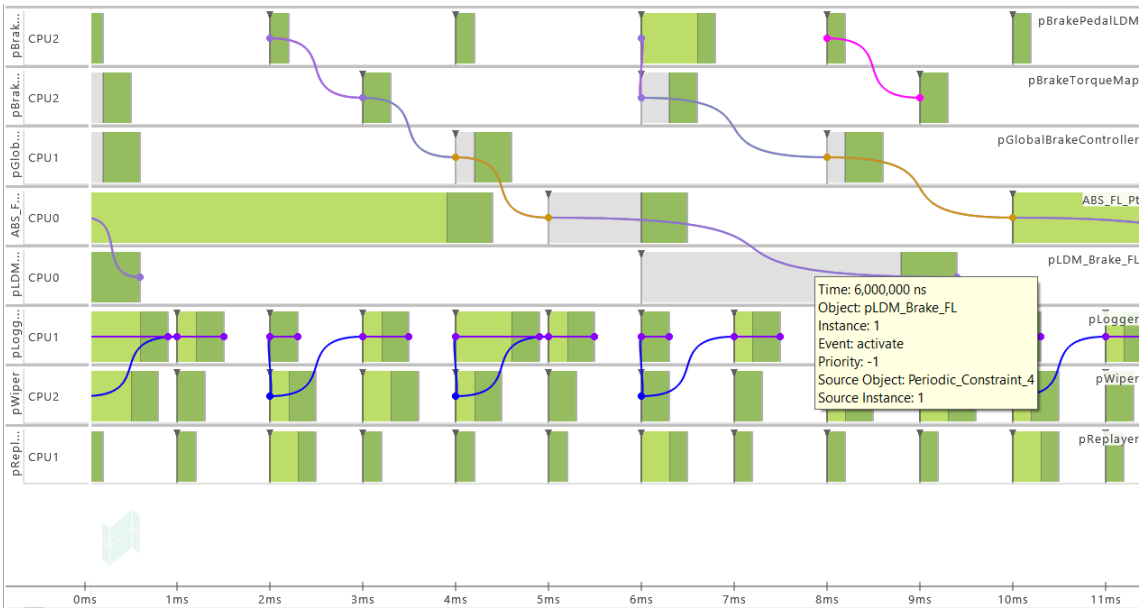


Figure 4.8: Simulation Grantt Chart

Validat...	Id	Name	Scope	Metric	Limit and Value	Fulfillment	Violati...	Count	Severity
✓		ABS_RL_Pt_Requirement	ABS_RL_Pt	Response Time	≤ 5.00000 ms	Fulfilled	0	401	Critical
✓		ResponseTime_UpperLimit_pRepl...	pReplayer	Response Time	≤ 1.00000 ms	Fulfilled	0	2001	Major
✓		ResponseTime_UpperLimit_pWip...	pWiper	Response Time	≤ 1.00000 ms	Fulfilled	0	2001	Major
✓		pBrakeTorqueMap_Requirement	pBrakeTorqueMap	Response Time	≤ 3.00000 ms	Fulfilled	0	667	Critical
✓		pLDM_Brake_RL_Requirement	pLDM_Brake_RL	Response Time	≤ 6.00000 ms	Fulfilled	0	334	Critical
✓		pLDM_Brake_RR_Requirement	pLDM_Brake_RR	Response Time	≤ 6.00000 ms	Fulfilled	0	334	Critical
✓		pLDM_Brake_FR_Requirement	pLDM_Brake_FR	Response Time	≤ 6.00000 ms	Fulfilled	0	334	Critical
✓		ABS_RR_Pt_Requirement	ABS_RR_Pt	Response Time	≤ 5.00000 ms	Fulfilled	0	401	Critical
✓		ResponseTime_UpperLimit_pLogg...	pLogger	Response Time	≤ 1.00000 ms	Fulfilled	0	2001	Major
✓		ABS_FR_Pt_Requirement	ABS_FR_Pt	Response Time	≤ 5.00000 ms	Fulfilled	0	401	Critical
✓		pLDM_Brake_FL_Requirement	pLDM_Brake_FL	Response Time	≤ 6.00000 ms	Fulfilled	0	334	Critical
✓		pBrakePedalLDM_Requirement	pBrakePedalLDM	Response Time	≤ 2.00000 ms	Fulfilled	0	1001	Critical
✓		ABS_FL_Pt_Requirement	ABS_FL_Pt	Response Time	≤ 5.00000 ms	Fulfilled	0	401	Critical
✓		pGlobalBrakeController_Require...	pGlobalBrakeCo...	Response Time	≤ 4.00000 ms	Fulfilled	0	501	Critical

Figure 4.9: Simulation Summary

5

Conclusion

5.1 Discussion

In the WCET estimation, since we do not have the hardware to be able to perform an accurate calculation, instead, we have to make assumptions as well as refer to other resources. For instance, we assume that our implementation is based on CAN network and also we assume that the rebuild of our logger will have minimal effect on the time consumption. Besides, it's a fact that the EE system on trucks is very a complex system consists of many different components. Therefore, if we work directly with such a system to design a new software component, it would be a huge workload for us to complete all validating, testing, building phases for the whole integration process of EE system on trucks. For the future research and implementation, it would be better to implement the design on the real hardware of trucks when we have more budget of time so that accurate WCET estimations can be accomplished.

We also understand that our virtual environment for evaluation, which consists of BBW, Wiper module, Logger and Replayer, is very specific rather than general. In practice, there may be more complex components in a system, and hence, our evaluation is not true for all those cases. However, we also showed an approach that can be used in case a software developer has to consider to design for different platforms in which one of them is real-time system. The design choices may vary yet the method stays unchanged.

In addition, a comparison of the impact of the choice of log file format is also a very interesting point to investigate for this project. It is the lack of time and tool supports that prevents us to focus more on this question. In fact, we got WCET result for two different formats which are CSV and custom format and they does affect to our designs performance in realtime analysis. The difference should have come from the format structure in our designs. However, we still cannot explain in details how and why the format structure affect to their corresponding WCET. Therefore, this drawback is also an important point if we have chance to continue this research.

Moreover, in many commercial software products for real-time systems, one can apply other advanced scheduling algorithm. For example, we can use OSEK [34] in TA Tool to simulate the schedulability for our tasks set. The results in APPENDIX A

showed that OSEK can also provide feasibility for our design as EDF does. Therefore, our design can be used in other systems with different scheduling algorithm as well.

As a result, we would provide our answers for research question (RQ) we made in Section 1.1 as follows:

- **RQ1**: By designing software component for Logger as shown in Section 4.2, we proposed an approach to collect internal values of signals at every millisecond. Our software design for Logger meets requirements of ADAPT and EE system on trucks
- **RQ2**: Similarly, we proposed a software design for Replayer where the logged data would be constructed in some file formats and the data is transferred into the system for each millisecond.
- **RQ3**: In order to evaluate our software designs in both ADAPT and EE system on trucks, we proposed a proof of concept for building a subsystem using the support of Amalthea and TA Suite tool as shown in Section 4.4. By this way, we proposed a new way of design any new software component for ADAPT and EE system on trucks.

5.2 Conclusion

This thesis project provides an approach of developing an applicable logging and replaying technology for both simulator and real-time systems on trucks. The part for the simulator is well-implemented and works properly with the ADAPT simulation environment. It includes a logger that can be configured to listen to a specific set of signals as well as generating log files in various format, and a replayer which uses the log file to reproduce the scenario at certain moment. The part for the real-time systems is done within simulation, we have estimated the timing cost of our logging technology and performed a hardware simulation that runs our software along with other truck software without timing violation. By doing this part, we provide a proof of concept of how the logging technology can be used in the truck EE systems. It can be used for future hardware level development of the logging technology.

Bibliography

[1] Data logging with the gl logger family. https://www.vector.com/int/en/products/products-a-z/hardware/gl-logger/. Accessed: 2019-03-30.

[2] Kaijser Henrik, Lönn Henrik, and Thorngren Peter. Virtual Integration on the Basis of a Structured System Modelling Approach. In *Proceedings of the Workshop CARS 2016 - The International Workshops on Critical Automotive Applications: Robustness & Safety*, Göteborg, Sweden, 10 2016.

[3] S. Abbott-McCune and L. A. Shay. Techniques in hacking and simulating a modem automotive controller area network. In *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*, pages 1–7, Oct 2016.

[4] Amalthea - An open platform project for embedded multicore systems. http://www.amalthea-project.org/. Accessed: 2019-03-30.

[5] J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, Oct 1988.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM, 1973*, 1973.

[7] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. Softw. Eng.*, 16(3):360–369, March 1990.

[8] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[9] AbsInt. ait worst-case execution time analyzers. https://www.absint.com/ait/index.htm/. Accessed: 2020-05-28.

[10] Tidorum Ltd. Bound-t time and stack analyzer. http://www.bound-t.com/. Accessed: 2020-05-28.

[11] National University of Singapore. Chronos. https://www.comp.nus.edu.sg/~rpembed/chronos/. Accessed: 2020-05-28.

[12] Autosar - Enabling continuous innovations. https://www.autosar.org/. Accessed: 2019-03-30.

[13] Autosar introduction. https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_Introduction.pdf. Accessed: 2019-03-30.

[14] F. Khenfri, K. Chaaban, and M. Chetto. A novel heuristic algorithm for mapping autosar runnables to tasks. In *2015 International Conference on Pervasive*

and Embedded Computing and Communication Systems (PECCS), pages 1–8, Feb 2015.

[15] Henrik Kaijser, Henrik Lönn, Peter Thorngren, Johan Ekberg, Maria Henningsson, and Mats Larsson. Towards simulation-based verification for continuous integration and delivery. In *2018 Embedded Real Time Software And Systems (ERTS)*, 04 2018.

[16] FMI development group: FMI 2.0. Functional mock-up interface (fmi). `www.fmi-standard.org`. Accessed: 2019-03-30.

[17] Fulvio Risso Loris Degioanni, Mario Baldi and Gianluca Varenni. Profiling and Optimization of Software-Based Network-Analysis Applications. In *Proceedings of the 15th IEEE Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)*, Sao Paulo, Brazil, 11 2003.

[18] M. Rojas-Fernández, D. Mújica-Vargas, M. Matuz-Cruz, and D. López-Borreguero. Performance comparison of 2D SLAM techniques available in ROS using a differential drive robot. In *2018 International Conference on Electronics, Communications and Computers (CONIELECOMP)*, pages 50–58, Feb 2018.

[19] ASAM MDF 2019. Mdf block. `https://www.asam.net/standards/detail/mdf/wiki/`. Accessed: 2019-10-30.

[20] Lowinski Martin, Ziegenbein Dirk, and Glesner Sabine. Partitioning embedded real-time control software based on communication dependencies. In *In proceedings of the International Workshop on Modelling in Automotive Software Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, Ottawa, Canada, September 2015.

[21] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel. Amalthea — Tailoring tools to projects in automotive software development. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 2, pages 515–520, Sep. 2015.

[22] C. Brink, E. Kamsties, M. Peters, and S. Sachweh. On hardware variability and the relation to software variability. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 352–355, Aug 2014.

[23] D. Fruhner, R. Höttger, S. Köpfer, and L. Krawczyk. Partitioning and mapping for embedded multicore system utilization in context of the model based open source development environment platform amalthea. In *Proceedings of the International Research Conference*, Dortmund, Germany, 2014.

[24] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-based automotive partitioning and mapping for embedded multicore systems. In *International Conference on Parallel, Distributed Systems and Software Engineering 2015*, Istanbul, Turkey, January 2015.

[25] R. Hoettger, B. Igel, and E. Kamsties. A novel partitioning and tracing approach for distributed systems based on vector clocks. In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, volume 02, pages 670–675, Sep. 2013.

[26] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within au-

tosar. In *2013 International Conference on Applied Electronics*, pages 1–6, Sep. 2013.

[27] A. Sailer, S. Schmidhuber, M. Hempe, M. Deubzer, and J. Mottok. Distributed multi-core development in the automotive domain - a practical comparison of ASAM MDX vs. AUTOSAR vs. AMALTHEA. In *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, pages 1–8, April 2016.

[28] Alan R. Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2), 2007.

[29] Pete Becker. Working draft, standard for programming language c++. page 797, 02 2011.

[30] Park and Byoungwook Choi. Design and implementation procedure for an advanced driver assistance system based on an open source autosar. *Electronics*, 8:1025, 09 2019.

[31] Infineon Technologies AG. Product brief: Aurix™- tc297t/tc298t/tc299t. `https://www.infineon.com/dgdl/Infineon-AURIX-TC297T_TC298T_TC299T-PB-v01_00-EN.pdf?fileId=5546d4625696ed76015697b2327f2460`. Accessed: 2019-10-30.

[32] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. In *Operations Research*, volume 26, pages 127–140, 1978.

[33] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 25–33, June 2000.

[34] J. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline an open source implementation of the osek/vdx rtos specification. In *2006 IEEE Conference on Emerging Technologies and Factory Automation*, pages 62–69, Sep. 2006.

[35] S. K. Baruah and Shun-Shii Lin. Pfair scheduling of generalized pinwheel task systems. *IEEE Transactions on Computers*, 47(7):812–816, July 1998.

A

Appendix A

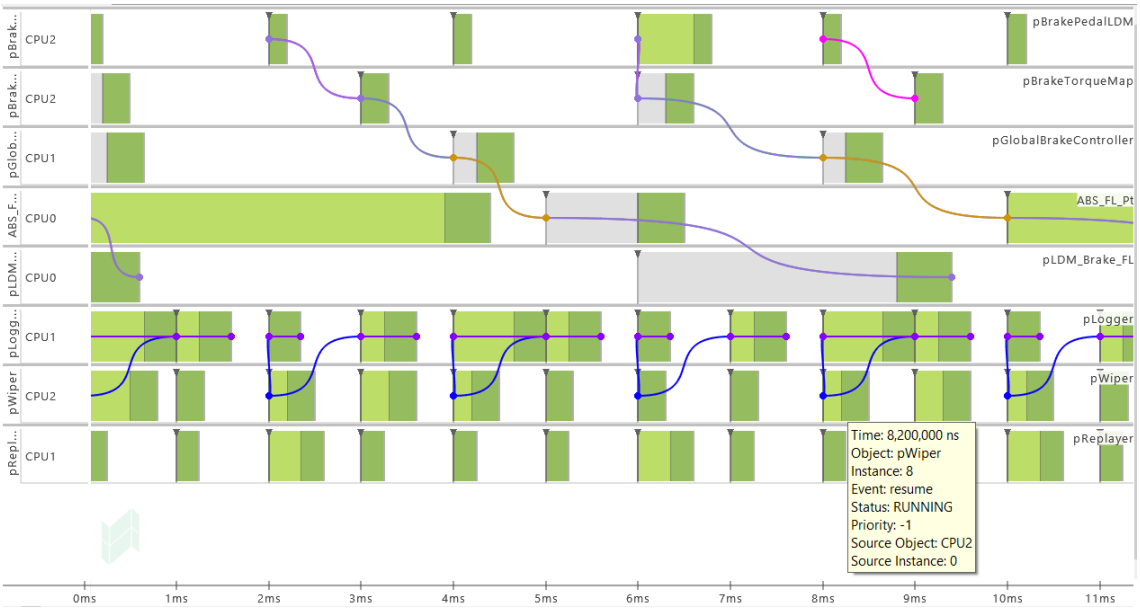


Figure A.1: Simulation Grantt Chart for maximum WCET variations with EDF

Name	Priority [co...	MTA_ERR count	Deadline [ms]	A2A avg [ms]	RT max [...	NRT max [ratio]	NET min [ms]	NET max [ms]
ABS_FL_Pt		0	5.00000	5.00000	4.40000	0.88000	0.50000	0.50000
ABS_FR_Pt		0	5.00000	5.00000	2.90000	0.58000	0.50000	0.50000
ABS_RL_Pt		0	5.00000	5.00000	2.40000	0.48000	0.50000	0.50000
ABS_RR_Pt		0	5.00000	5.00000	3.30000	0.66000	0.50000	0.50000
pBrakePedalLDM		0	2.00000	2.00000	0.80000	0.40000	0.20000	0.20000
pBrakeTorqueMap		0	3.00000	3.00000	0.60000	0.20000	0.30000	0.30000
pGlobalBrakeContr...		0	4.00000	4.00000	0.65000	0.16250	0.40000	0.40000
pLDM_Brake_FL		0	6.00000	6.00000	3.90000	0.65000	0.60000	0.60000
pLDM_Brake_FR		0	6.00000	6.00000	2.80000	0.46667	0.60000	0.60000
pLDM_Brake_RL		0	6.00000	6.00000	2.80000	0.46667	0.60000	0.60000
pLDM_Brake_RR		0	6.00000	6.00000	3.90000	0.65000	0.60000	0.60000
pLogger		0	1.00000	1.00000	1.00000	1.00000	0.35000	0.35000
pReplayer		0	1.00000	1.00000	0.60000	0.60000	0.25000	0.25000
pWiper		0	1.00000	1.00000	0.80000	0.80000	0.30000	0.30000

Figure A.2: Simulation Summary for maximum WCET variations with EDF

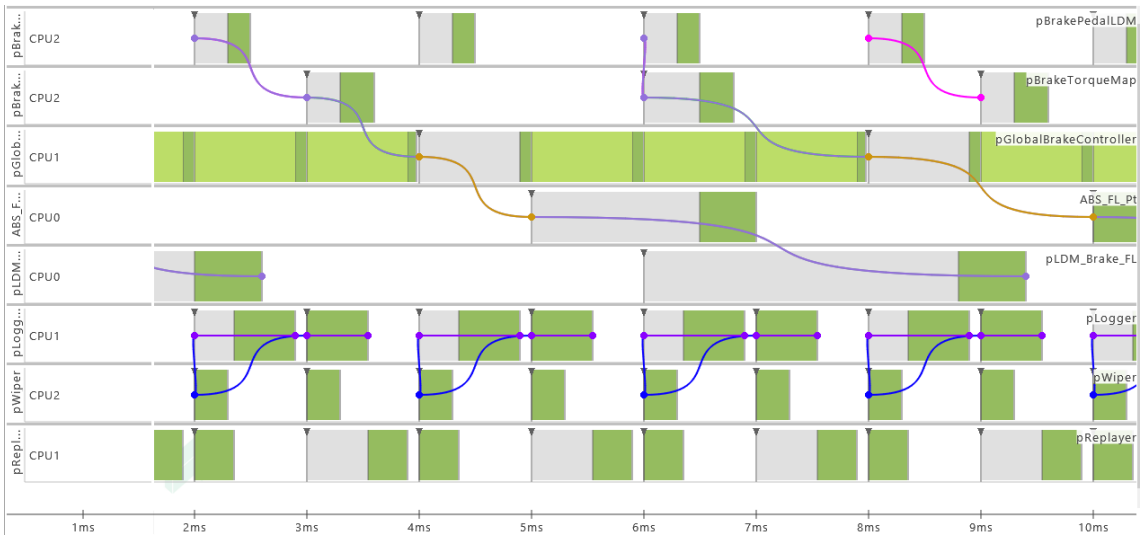


Figure A.3: Simulation Grantt Chart with OSEK

Name	Priority [co...	MTA_ERR count	Deadline [ms]	A2A avg [ms]	RT max [...	NRT max [ratio]	NET min [ms]	NET max [ms]
ABS_FL_Pt	1	0	5.00000	5.00000	3.40000	0.68000	0.50000	0.50000
ABS_FR_Pt	1	0	5.00000	5.00000	2.90000	0.58000	0.50000	0.50000
ABS_RL_Pt	1	0	5.00000	5.00000	2.40000	0.48000	0.50000	0.50000
ABS_RR_Pt	1	0	5.00000	5.00000	2.40000	0.48000	0.50000	0.50000
pBrakePedalLDM	1	0	2.00000	2.00000	0.50000	0.25000	0.20000	0.20000
pBrakeTorqueMap	1	0	3.00000	3.00000	0.80000	0.26667	0.30000	0.30000
pGlobalBrakeContr...	1	0	4.00000	4.00000	3.98000	0.99500	0.40000	0.40000
pLDM_Brake_FL	1	0	6		4.40000	0.73333	0.60000	0.60000
pLDM_Brake_FR	1	0	6		3.80000	0.63333	0.60000	0.60000
pLDM_Brake_RL	1	0	6		3.80000	0.63333	0.60000	0.60000
pLDM_Brake_RR	1	0	6		4.40000	0.73333	0.60000	0.60000
pLogger	2	0	1.00000	1.00000	0.89500	0.89500	0.54500	0.54500
pReplayer	2	0	1.00000	1.00000	0.89500	0.89500	0.35000	0.35000
pWiper	2	0	1.00000	1.00000	0.30000	0.30000	0.30000	0.30000

4.000000 ms

Base value: 4.000000 ms

Figure A.4: Simulation Summary with OSEK