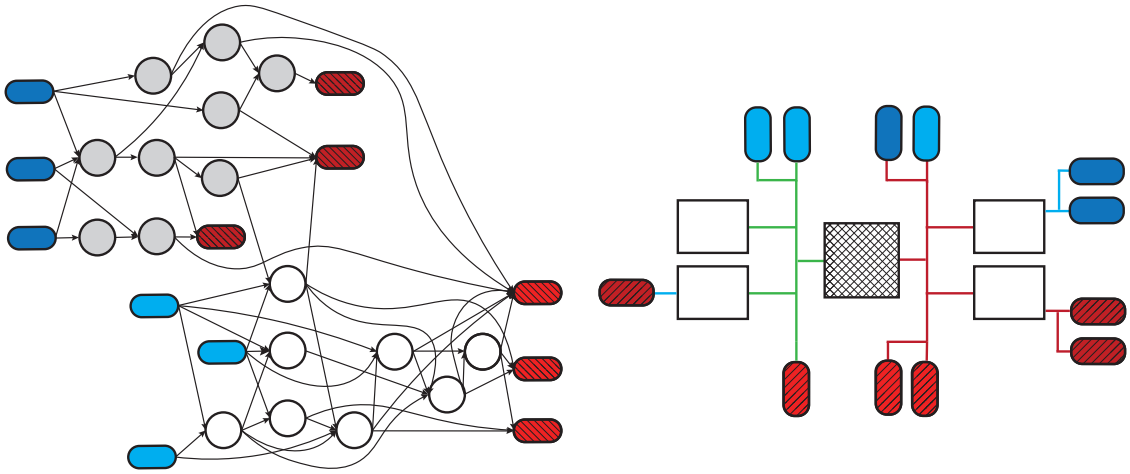
CHALMERS
UNIVERSITY OF TECHNOLOGY



Optimization of intra-vehicle architecture using a multi-objective genetic algorithm

Master's thesis in Complex Adaptive Systems

ALBIN LORENTZSON
VIKTOR TENGNÄS

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017:50

Optimization of intra-vehicle architecture using a multi-objective genetic algorithm

ALBIN LORENTZSON & VIKTOR TENGNÄS

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Adaptive Systems Group
Chalmers University of Technology
Gothenburg, Sweden 2017

Optimization of intra-vehicle architecture using a multi-objective genetic algorithm
ALBIN LORENTZSON & VIKTOR TENGNÄS

Supervisor & examiner: Mattias Wahde, Department of Applied Mechanics

Master's Thesis 2017:50
ISSN: 1652-8557
Department of Applied Mechanics
Division of Vehicle Engineering and Autonomous Systems
Adaptive Systems Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Small-scale automotive architecture generated by developed framework.

Typeset in LaTeX
Gothenburg, Sweden 2017

Optimization of intra-vehicle architecture using a multi-objective genetic algorithm
ALBIN LORENTZSON & VIKTOR TENGNÄS
Department of Applied Mechanics
Chalmers University of Technology

Abstract

Using problem-specific genetic operators, the multi-objective genetic algorithm Non-dominated sorting genetic algorithm II (NSGA-II) is adapted to the allocation of software components (SWCs) to electric control units (ECUs) within an automotive architecture. A simulation environment is developed in order to assess the performance of the allocation within an architecture, and thereby provide the genetic algorithm with objective and constraint values. The validity of the optimization method is evaluated by generating artificial software and hardware architectures, and allowing the genetic algorithm to optimize the software allocation. A novel algorithm for routing signals within the software architecture, based on forming and connecting vehicle features, is presented.

The optimized Pareto-fronts of small-scale (17 SWCs and 4 ECUs) automotive architectures are compared to the ground truth through exhaustive search. The average hypervolume ratio is 98.9%, computed over 10 architectures and 100 optimizations, and 48% of the performed optimizations successfully found the entire true Pareto-front. For the large-scale (250 SWCs and 25 ECUs) architectures, no ground truth can obtained, and the optimizations are instead evaluated with regard to consistency. In general, the optimization method quickly finds feasible solutions. However, discrepancies between the approximated Pareto-fronts suggest that premature convergence sometimes occurs.

Even though the results indicate that the optimization method works as intended and yields satisfactory results with respect to formulated aims, it is not evident that this method is applicable to the optimization of real automotive architectures. The true nature of these architectures may be too complicated to be able to be compressed into a feasibly low number of objectives, which the developed optimization method requires to perform well.

Keywords: Automotive architecture, software component, electronic control unit, software allocation, multi-objective optimization, NSGA-II, busload, memory utilization, signal delivery time, feature generation

Acknowledgements

First of all, we would like to thank Prof. Mattias Wahde, our supervisor at Chalmers, for his support and assistance throughout this project. You have been a great source of inspiration during the past years.

Many thanks also go to Elpidoforos Arapantonis and Lena Eliasson, our supervisors at Volvo Car Corporation, for always taking time to answer our questions.

Finally, we would like to express our gratitude to all other friendly people at Volvo Car Corporation and our friends at Chalmers.

Albin Lorentzson & Viktor Tengnäs, Gothenburg, June 2017

Contents

1
Introduction

In this chapter, a brief background to software allocation, together with some of its challenges, are presented. Thereafter, the description of the problem treated in this report is summarized, including some of the main aspects needed to be taken into consideration when allocating software. The chapter continues with the aim and scope of the thesis, and ends with an overview of the remaining chapters.

1.1 Background

The automotive industry of today differs a great deal from what it looked like a few decades ago [1]. With increased demands for safety, comfort, and infotainment, as well as requirements for reduced environmental impact, a major part of today's innovation is made in regard to electronics and software [2]. These developments have resulted in extensive hardware networks of electronic control units (ECUs), sensors, and actuators. ECUs are embedded systems of microcontrollers and memories, which contain the software, perform computations, and control actuators [3].

In order to enable software within different hardware components to exchange signals, the ECUs, sensors, and actuators must be connected. Previously, these connections were established by drawing a cable between two hardware components that needed to communicate [4]. As the amount of hardware and software in a vehicle increased, this approach was deemed insufficient. Specialized communication protocols called buses, capable of transmitting multiple signals between several different ECUs, were thus developed. The use of communication buses provides both a better overview and a cost reduction, and also facilitates the assurance of for example message delivery.

Which signals are transmitted over a certain bus depend on the topology of the hardware network, as well as on how the software is distributed over the set of ECUs. During the past decade, middleware concepts, such as AUTOSAR [5], has enabled freedom in the allocation of software to ECUs, by abstracting the hardware and software. This is made possible through the definition of a software component (SWC), which is a unit of software that performs an application or a part of an application. By including each SWC with basic software for memory management, diagnostics services, bus communication, etc., an SWC may be assigned to a wider range of ECUs.

With the increased freedom in the allocation of software to hardware, the problem of making an efficient allocation arises. A simple approach to software allocation is to place all the software used for a certain vehicle functionality on the same ECU, and then connecting that ECU to a bus close to the corresponding sensors and actuators [6]. Such allocations are able to reduce the amount of wiring in the hardware network, but result in a costly proliferation of ECUs. A more cost-efficient approach is to optimize the software allocation with respect to the existing hardware network.

1.2 Problem description

The problem of optimizing the allocation of SWCs to ECUs can be regarded as an extension of the classic bin-packing problem [7], i.e. the problem of allocating a set of objects with certain sizes to a set of containers in a way that minimizes the number of containers used. This NP-complete problem [7] considers object sizes and container capacities, which correspond to the sizes or processing demands of SWCs and the corresponding capacities of ECUs. The problem extension is due to restrictions on the signal exchange between SWCs,

limitations on the communication capabilities between ECUs, etc. These additional aspects of the automotive architecture, a term denoting both the hardware and the software, need to be considered when allocating software. Some of them are summarized in the following list:

- **ECU memory.** An ECU has a limited amount of available memory, which restricts the amount of software that may be allocated to it [8].
- **ECU processing power.** The processing power of an ECU must be greater than the processing power demands of the allocated SWCs [9].
- **Busload.** A communication bus has limitations in regard to the data transmission rate, which limits the number of signals it can transmit per time unit [4].
- **Manufacturing cost.** ECUs without any allocated SWCs may be removed from the hardware network, which lowers the production cost [4].
- **Hardware demand.** An SWCs may require the presence of certain hardware. This could mean that some ECUs are not suitable for allocation of that SWC [3].
- **Co-location.** Two SWCs may require that both are allocated to the same ECU, or that they must be allocated to different ECUs [3].
- **Signal delivery time.** Signals may be time-critical and require that they are received before a certain amount of time has passed [4].

Performing the allocation becomes even harder with many of these parameters conflicting with each other, making trade-offs necessary when optimizing. For example, SWCs that communicate frequently should ideally be placed within the same ECU, in order to decrease the busload of the network [10]. However, at the same time the maximum amount of available memory of the ECU cannot be exceeded. Since neither memory utilization nor busload may be regarded as more important than the other, one approach is to handle the allocation of software as a multi-objective optimization problem. This type of optimization problem is defined in greater detail in Section 4.1, where also the concept of Pareto-optimality is discussed, as it is required for comparing the relative merits of different potential solutions to multi-objective optimization problems.

1.3 Aim

The main aim of the work presented here is to develop and implement a multi-objective genetic algorithm for optimization of the software allocation in an automotive architecture. In order to fulfill this purpose, this thesis focuses on the following three tasks:

- Development of a simulation environment for representation of automotive hardware and software, as well as evaluation of a software allocation used in that architecture.
- Implementation of a multi-objective genetic algorithm for optimization of the software allocation used in an automotive architecture.
- Development of a framework for generation of realistic automotive architectures, used to evaluate the validity of the implemented algorithm.

The mentioned simulation environment and optimization method could facilitate the software allocation at Volvo Cars, thus improving the safety and efficiency of the company's vehicles.

1.4 Scope and limitations

In order to assess the performance of a software allocation, the parameters and dynamics of relevant entities within automotive architectures were modeled in a simulation environment using C# .NET. The evaluation was made in regard to a collection of predefined fitness measures and constraints. Of the parameters described in Section 1.2, busload and memory utilization were used as both objectives and constraints, while signal delivery time, hardware demand, and co-location were solely implemented as constraints.

For the optimization task, the performance of the automotive architecture could only be influenced through modification of the allocation of software, i.e. the functional descriptions and hardware networks were assumed to be set. Explicitly, this implies that that no changes could be made in regard to the communications between SWCs during the optimization. Additionally, buses could not be rerouted and the set of ECUs could not be altered.

The implementation of the optimization algorithm was made in C# .NET and started from already existing metaheuristics for multi-objective genetic algorithms. Certain aspects of the algorithm, such as the allocation encoding and the genetic operators, were adapted to the problem of software allocation.

In order to assess the potential and limitations of the implemented multi-objective genetic algorithm, it was applied to a wide range of automotive architectures. These architectures were generated through a developed framework, which produces software descriptions and virtual hardware networks that resemble the architectures used by Volvo Cars. The framework is capable of generating architectures of varying sizes and complexities through a stochastic procedure. The stochastic nature makes it difficult to guarantee that, for a generated architecture, there exists a software allocation which satisfies all constraints. Precautions, such as dynamically assigning certain generation parameters, have been taken in order to prevent some causes of infeasibility.

Validating the algorithm through the developed framework, rather than through an industrial case study, allows for a more comprehensive assessment of its validity. On the other hand, an industrial case study has an inherent allocation that an optimized allocation may be compared to. The developed framework provides no such counterpart.

1.5 Outline of report

To further acquaint the reader with the concept of automotive architecture, Chapter 2 gives a more detailed description of the hardware and software relevant for this thesis, with particular emphasis on the communication protocols used in vehicles. Familiarity with these protocols is essential for understanding the models for busload and signal delivery time. Chapters 3 to 5 follow with presentations of the methods used to solve the three tasks presented in Section 1.3. This includes in-depth descriptions of the models for each objective and constraint, the Non-dominated Sorting Genetic Algorithm II, and the developed algorithm for generating realistic software architectures.

The results of this thesis, which consists of studies of the optimization of both small-scale and large-scale automotive architectures, generated by the developed framework, are presented in Chapter 6. These results are discussed in Chapter 7, together with some thoughts regarding the validity of applying this work for industrial usage. Chapter 8 concludes the thesis by stating the main results and discussion topics.

2
Automotive architecture

This chapter covers some basics of automotive architecture. The content has deliberately been simplified to only include the information needed to understand the models described in Chapters 3 and 5. Subjects outside the scope of this thesis, such as the dynamics inside ECUs, have been left out. The first section provides a introduction to AUTOSAR, and is followed by high-level information about the hardware and software architecture of vehicles. How software and hardware are combined, in order to create functionalities in the vehicle, is thereafter presented. Lastly, more in-depth explanations of three commonly used bus protocols, CAN, LIN, and FlexRay, are given.

2.1 The middleware concept of AUTOSAR

As mentioned in Section 1.1, the hardware and software of a modern vehicle may to some extent be regarded as separate architectures [10]. Standardization through the development of middleware concepts has been essential in achieving this abstraction. One widely used middleware concept is AUTomotive Open System ARchitecture (AUTOSAR) [5], which is the result of a major cooperation between car manufacturers, suppliers, and software companies within the automotive industry. Notable partners include BMW, Toyota, IBM, and Volvo Cars [11].

AUTOSAR's most important features include increased scalability and modularity for automotive architectures, partly through allowing SWCs to be transferred between different hardware platforms [4, 8]. This is accomplished through the definitions of common interfaces that enable the SWCs to exchange signals using the middleware and hardware abstraction layers, omitting direct contact with the hardware and its underlying operating systems [8], something that is illustrated in Figure 2.1. The signal between SWC 1 and SWC 3 is an example of inter-ECU communication, since the two SWCs are allocated to different ECUs and the signal thus must traverse the bus that connects the two ECU. In this case, ECU 1 is said to transmit this signal to ECU 2, even though the actual sender and receiver are the two mentioned SWCs.

The introduction of AUTOSAR has greatly facilitated faster progress within the automotive industry. Manufacturers are able to choose more freely which functions to develop within their company and which to order from suppliers. Furthermore, suppliers do not need to completely alter their products to accommodate new customers' existing architectures, and may instead focus on further function development. This increases the competition among suppliers and widens the range of alternatives that manufacturers can choose from.

2.2 Software architecture

The software architecture is embodied by SWCs and the signals that they exchange. An SWC's size depends on which task it performs and ultimately the amount of program code written to implement the task. Each SWC receives input signals, processes these according to its implemented task, and then transmits its output to one or several receiving SWCs. Its output can be either a single signal, e.g. the engine temperature, or a signal group, e.g. the wheel speeds [5]. In the latter case, a receiver of the signal group does not necessarily use every constituent signal. Instead, the receiver only uses the signals required as input for its task.

In the context of automotive architecture, the term *domain* is used for grouping hardware and software components that fall under the same category of functionality. The task performed by an SWC determines the domain to which it belongs. AUTOSAR recognizes six different domains: safety, chassis, powertrain,

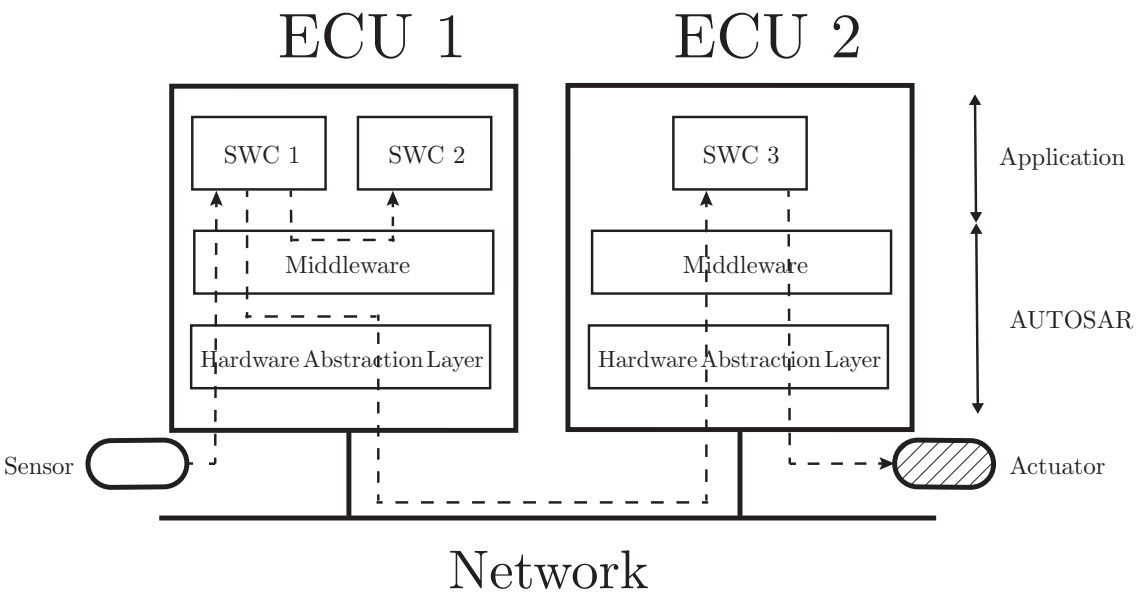


Figure 2.1: ***Intra- and inter-ECU communication enabled by AUTOSAR.*** *SWCs allocated to the same ECU may exchange signals directly via the middleware. However, inter-ECU signals and signals sent from sensors or to actuators must also be transmitted through the hardware abstraction layer. The figure is based on [4].*

body and comfort, multimedia and telematics, and man-machine-interface [11]. As an example, SWCs in the powertrain domain manage engine and transmission.

The task of an SWC also affects the properties of its output signal, for example in regard to time-criticality. Signals may have specific end-to-end (E2E) times, which are the longest acceptable delivery times for those signals. The delivery time is computed as the time between the signal being initiated by the sender and acquired by the receiver, and is mainly affected by the signal's size and transmission through the hardware network. Two different types of signals exist, periodic and sporadic. The value of a periodic signal is updated repeatedly with a static updating time, while the initiation of a sporadic signal is triggered by an event, for example the deployment of the airbags. Consecutive initiations of a sporadic signal are though limited by a minimum time separation [12].

2.3 Hardware architecture

Three important categories of hardware components are the ECUs, sensors, and actuators. These are connected via buses and gateways, which govern their communication [13]. An example of such connections, in the form of a small-scale hardware network, can be seen in Figure 2.2. The main computing components in the network are the ECUs, marked as rectangles in the figure. These units are often developed to suit certain needs and will therefore differ with respect to properties such as processing speed, memory capacity, physical size, cost, and safety level [3, 4, 9].

The sensors connected to the hardware network, which comprise everything from switches to cameras, are the main sources of input to the software architecture [4]. Some sensor signals can be initiated by a user of the vehicle, such as pressing the gas pedal, while others provide data without manual interaction, e.g. measuring the engine temperature or wheel speeds. Actuators are used in order to execute the mechanical or electrical commands in a vehicle, such as rolling down a window or emergency braking.

There are physical limitations when choosing the positions of the sensors and actuators in a vehicle. Many of these hardware components need to be placed in proximity of what should be measured or executed, but factors such as vibration must also be taken into consideration [15]. These restrictions have consequences for the rest of the hardware network, since the physical placement of ECUs relative to sensors and actuators affect aspects such as the number of buses that needs to be used as well as their wiring. Because of this,

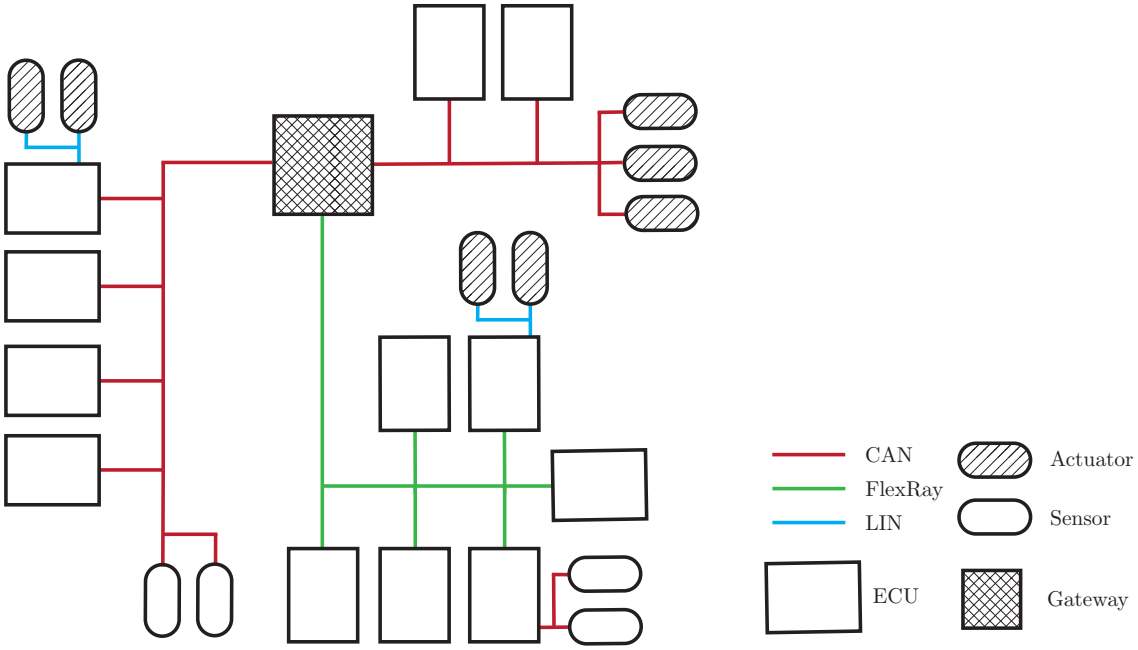


Figure 2.2: ***A small-scaled hardware architecture.*** *Each ECU is connected to one of the three subnetworks, which consist of two CAN buses and one FlexRay bus. These three subnetworks are in turn joined by a gateway. Sensors and actuators connect either directly to one of the subnetworks, or to an ECU via a LIN or CAN bus. The figure is made with inspiration from [9, 14].*

some sensors and actuators are directly connected to several ECUs and others are only directly connected to a single ECU, see Figure 2.2.

2.4 Combining hardware and software into vehicle features

Together with sensors and actuators, SWCs are used to implement functionalities, or features, into a vehicle [9]. An example of such a feature is the key-less entry function, which allows the user to unlock the vehicle without bringing out the key. A schematic of this feature, inspired by [9], can be seen in Figure 2.3. If any antenna detects the key in proximity of the vehicle, a signal is transmitted to the key entry control, which in turn sends a signal to the central door locking control. This SWC unlocks the door and transmits a signal to the SWC for direction indication, in order to show that the doors have been unlocked.

Figure 2.4 shows a possible allocation of the key-less entry feature. In this example, the hardware configuration consists of two ECUs connected by a bus, as well as the sensors and actuators used within the feature. Two of the SWCs are allocated to the same ECU and therefore the signal between these does not affect the busload of the bus. The last SWC is allocated to the second ECU and since the signal received by this SWC is transmitted over the bus, the busload is increased. Even this small example, with only one feature and two ECUs, illustrates the complexity of allocation, since there are already $2^3 = 8$ ways to allocate the three SWCs. An increase in the number of SWCs would cause the search space to expand exponentially.

2.5 Bus protocols

There does not exist a single all-purpose bus protocol within the automotive industry, but rather many different protocols that all have specific strengths and weaknesses. Three of these protocols are Local Interconnect Network (LIN) [16], Controller Area Network (CAN) [17], and FlexRay [18]. Characteristics and common applications of these bus protocols are presented in Table 2.1, and Subsections 2.5.1 to 2.5.3 describe each protocol in more detail.

Despite differing in several regards, LIN, CAN, and FlexRay share some properties. Each bus protocol has

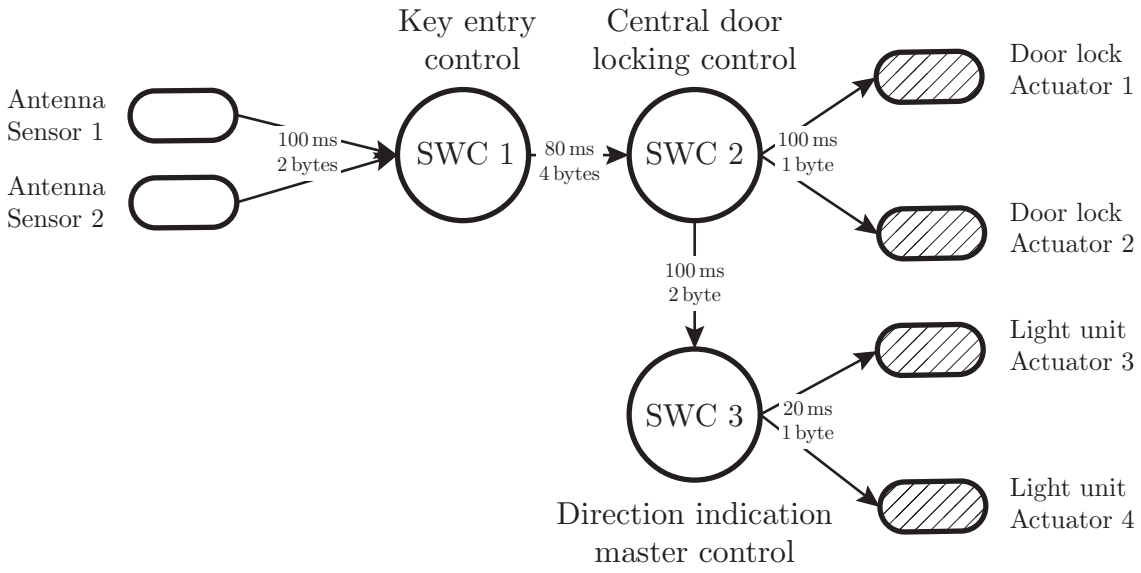


Figure 2.3: ***The hardware and software included in the key-less entry functionality.*** *The transmitted signals, together with their respective end-to-end times and data sizes, are illustrated as arrows between the actuators, SWCs, and sensors. The shown example is a simplification of the one used in [9].*

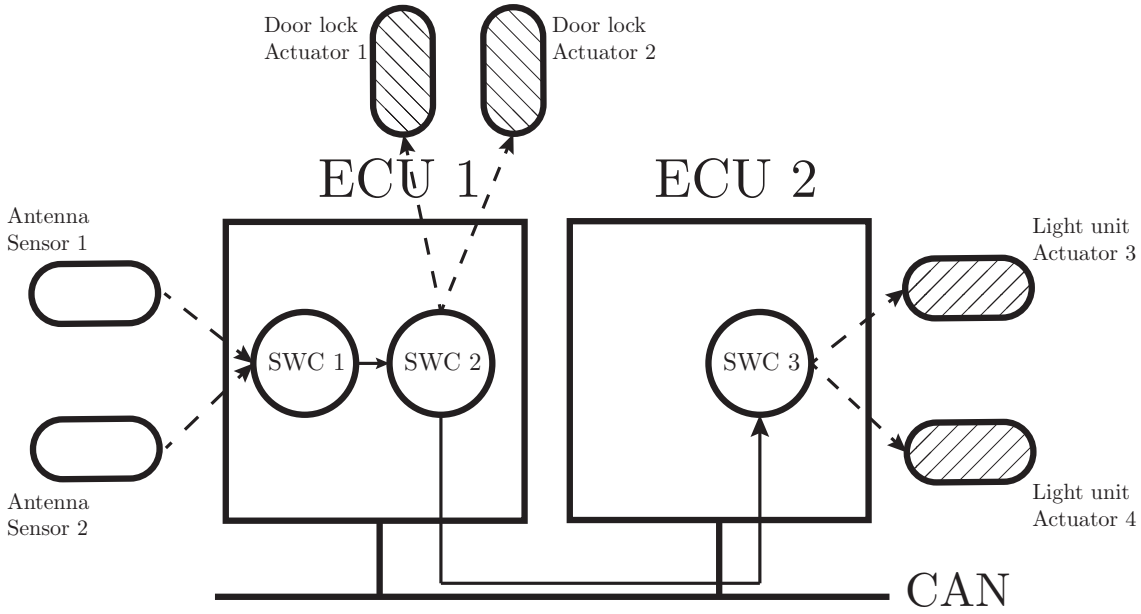


Figure 2.4: ***A possible allocation of the SWCs in the key-less entry functionality.*** *The hardware consists of two ECUs connected by one CAN bus, together with the sensors and actuators involved in the functionality. Note that the signal sent from SWC 2 to SWC 3 needs to be transmitted over the CAN bus, in contrast to the signal from SWC 1 to SWC 2. Since the signals transmitted between SWCs are the focus of this figure, whereas the locations of the sensors and actuators are not of importance, the signals sent from sensors or to actuators are illustrated by dashed lines.*

Table 2.1: ***Properties of the bus protocols LIN, CAN, and FlexRay.***

Bus	LIN	CAN	FlexRay
Bandwidth [4]	19.2 kb/s	500 kb/s	10 Mb/s
Cost [19]	Low	Medium	High
Messaging [19]	Deterministic, static scheduling	Event-triggered	Event- and time-triggered
Node control [19]	Master-Slave	Autonomous	Master-Slave, Autonomous
Typical applications [20]	Body electronics, e.g. mirrors and seats	Mainstream powertrain, e.g. engine and transmission	Safety and high-performance powertrain, e.g. adaptive cruise control

a certain bandwidth, or bit rate, with which it may transmit data [4]. From this general bus property, one can compute the busload, i.e. the bandwidth consumed by the transmitted signals divided by the available bandwidth. Additionally, every bus protocol uses the concept of frames for the transmission of signals. A frame is a sequence of bit values that together encode the data of signals transmitted over the bus. In order for the signal receivers to be able to correctly interpret the incoming information, each frame also contains so called overhead information. This includes a unique identification sequence that is associated with the specific signal composition and a data sequence that functions as error-detecting code. The data sequence containing the signal values is denoted the payload of the frame. In order to compute the transmission time of a frame, the number of bits in the frame is divided by the bandwidth of the bus.

2.5.1 Controller Area Network (CAN)

Out of the three bus protocols treated in this thesis, CAN was the first to be developed [21]. Its purpose was to replace the previous standard of point-to-point wiring between the hardware nodes of the vehicle, where hardware nodes denote the combined sets of ECUs, sensors, and actuators [21]. Figure 2.5 illustrates the format of a CAN frame as well as the effect of bit stuffing, i.e. the insertion of a bit of opposite value directly after five consecutive bits of the same value have been transmitted [22]. Bit stuffing is included in the CAN protocol both to maintain time synchronization between the connected nodes and as an error-detecting measure. The possibility of bit stuffing means that the delivery time for an arbitrary CAN frame is not constant, since it depends on the content of the transmitted signal.

CAN uses event-triggered scheduling, which means that there is no predetermined order in which the signals are transmitted. Instead, each signal has a unique priority, defined by its identifier segment. The next signal to be broadcast is determined through the process of arbitration, with respect to the signal priorities [17, 18]. Once the previous transmission has ended and the bus is free, any node connected to the bus may start the process of transmitting a signal, while simultaneously monitoring whether any other signals are being broadcast over the bus. If a node detects the transmission of a signal with higher priority than the one broadcast by itself, the node in question will terminate its transmission and reattempt to broadcast once the bus is free.

Through the event-triggered scheduling of signals, the CAN protocol ensures that outdated signal values are never retransmitted, which may occur for LIN and FlexRay. CAN buses thus have more efficient bandwidth utilization. However, the latency of a signal, i.e. the delay between the instruction to transmit the signal and its actual transmission, is indeterminate due to the arbitration [23]. The signal latencies increase with a higher busload, especially for the lower priority signals, since the bus is occupied more frequently.

During the first years after the introduction of CAN, low busloads ($30\% - 40\%$) and extensive testing were required in order to establish confidence that signals experienced acceptable delivery times over CAN [14, 23, 24]. When Tindell et al. [12] published a worst-case analysis, based on fixed priority non-preemptive

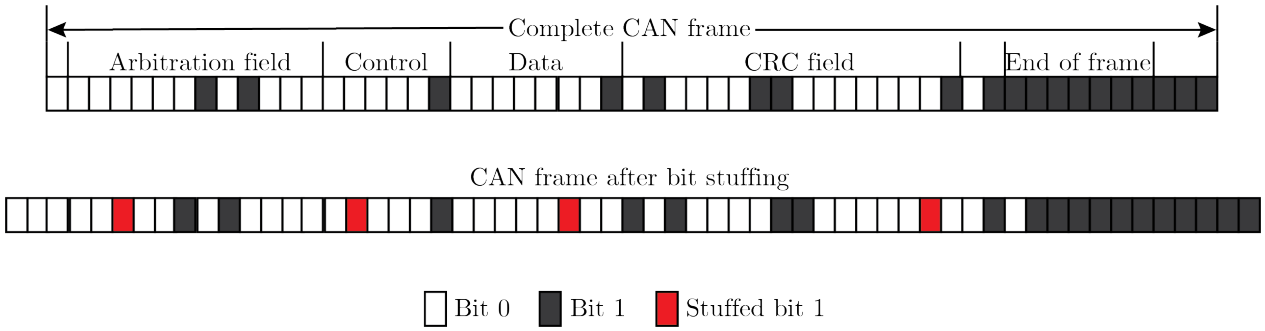


Figure 2.5: ***A simplified view of the format of a CAN frame, and the effect of bit stuffing.*** *The arbitration field contains the identifier sequence and the control indicates the length of the following data segment, which may be* $\{1, \ldots, 8\}$ byte. *The cyclic redundancy check (CRC) is used for error detection. The dark grey rectangles represent the bit value 1 and the white rectangles represent the bit value 0. The red rectangles in the bottom row represent stuffed bits. A bit is stuffed into the frame after each occurrence of five consecutive bits with the same value. Note that bit stuffing does not apply to the end of frame segment. The figure is a remake of [25], used under the CC BY-SA 3.0 licence.*

scheduling, manufacturers such as Volvo Cars were able to increase the busload to 80% while still ensuring acceptable delivery times [14]. This analysis has since been revisited by Davis et al. [14], to include the possibility of having signals with E2E times longer than their updating times.

2.5.2 Local Interconnect Network (LIN)

The LIN protocol was introduced as a cheaper alternative to the CAN bus. The reason behind its introduction was the fact that many applications within a vehicle, such as connecting simple sensors and actuators, do not require the high bandwidth and versatility that the CAN protocol offers [26]. The scheduling of signals on LIN is mainly time-triggered and a master-slave approach, involving one master node and several slave nodes, is used to enforce the broadcasting schedule. The master has one or several so called schedule tables, which specify when each slave is supposed to transmit. A slave may continuously update its respective signal values, but will only transmit once prompted by the master. By adhering to the schedule tables, a busload below 100% can be guaranteed, as can the periodicity for signal transmissions. The latter implies that the time between two consecutive transmissions of a signal is constant.

Each slave node in a LIN network is equipped with a simple oscillator that functions as an internal clock [16]. These do however not have high requirements for accuracy, which impairs the synchronization between the nodes [4]. In order to deal with this, each frame transmitted over LIN is dedicated a time budget 40% larger than what the amount of data would require with perfect synchronization, which implies longer transmission times for signals and a higher busload on the LIN bus.

2.5.3 FlexRay

While the LIN protocol provided a cheaper alternative to CAN, the aim when designing FlexRay was to develop a faster and more reliable bus protocol than CAN [20]. The FlexRay protocol is able to provide guarantees for signal delivery times and has a significantly increased bandwidth compared to CAN. These features make FlexRay suitable for safety-critical applications and operations with hard real-time requirements [19, 20].

Figure 2.6 illustrates the format of a FlexRay communication cycle, which is typically between 1 ms and 5 ms [20]. The static and dynamic segments are used for broadcasting signals, using time-triggered and event-triggered messaging respectively, the symbol windows is generally used for network maintenance, and the network idle time is used for synchronization [20]. Both the broadcasting segments are composed of slots, which essentially correspond to frames. Each slot in the static segment is dedicated to a specific ECU, and provides that ECU with an opportunity to broadcast signals each cycle. This means that the shortest possible period time for a signal, meaning how often it is broadcast on the FlexRay bus, is equal to the time

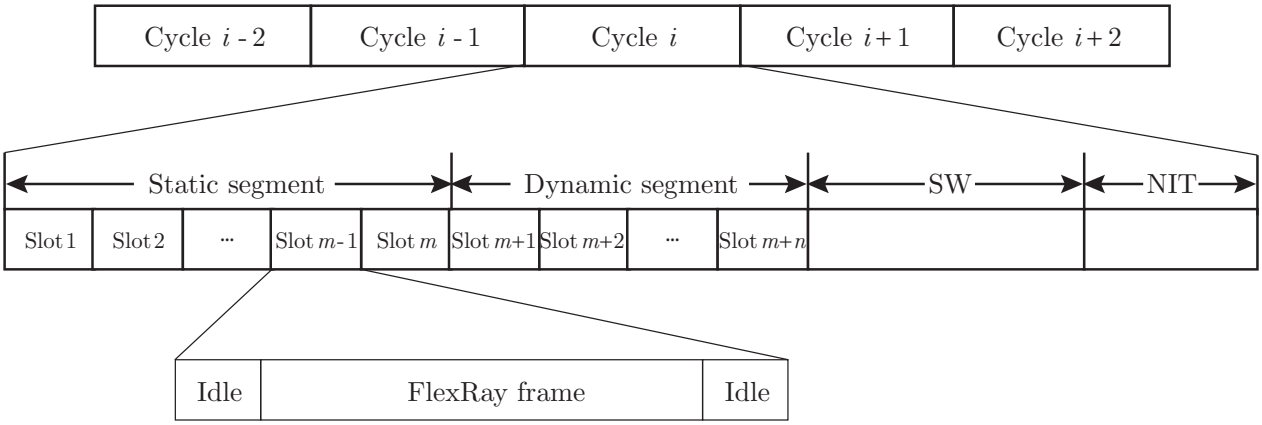


Figure 2.6: ***A schematic of the format of a FlexRay communication cycle.*** *Each cycle consists of a static segment, a dynamic segment, a symbol window (SW) and a network idle time (NIT). The slots contained within the static and dynamic segments are used for broadcasting signals over the bus. Note that the number of slots in each segment, n and m, depends on the configuration of the FlexRay bus. The FlexRay frame within a slot generally contain several different signal values.*

length of a communication cycle. However, a signal does not have to be transmitted every cycle, but may for example be assigned to every second or third cycle [27]. This is useful for signals with longer updating times, since their busload consumption may be decreased. The FlexRay protocol limits the number of different configurations to 64, which means that the same cycle configuration is repeated at least every 64^{th} cycle [27]. However, some vehicle manufacturers use fewer configurations.

3
Simulation Environment

This chapter describes the models used for evaluating an automotive architecture. Many of these models are taken from previous work [3, 4, 8, 14] and have been combined to form the simulation environment. Some preparation steps are required before the evaluation is possible, and these are therefore presented before the models are explained. For this work, the objectives considered are computed from the busloads and the memory utilization of the ECUs. These two aspects are also used to formulate constraints, together with the delivery times of signals, hardware demands for SWCs and co-location of SWCs. Each objective and constraint model yields a single value, representative of the model's assessment of the automotive architecture. These values are the ones used within the optimization procedure, described in Chapter 4.

3.1 Pre-evaluation procedure

As mentioned in Section 1.4, the simulation environment was created in order to evaluate the performance of an automotive architecture, or more specifically the software allocation applied to that architecture. Figure 3.1 illustrates the procedure that leads up to the evaluation of the the automotive architecture. As shown, the simulation environment requires complete descriptions of the hardware and software architectures, as well as the allocation of the SWCs to the ECUs. This means that both the connections between hardware components and the signal exchange between SWCs are known. These descriptions do however not contain information about how the inter-ECU signals should be transmitted over the hardware architecture. The routing of signals through buses, gateways, and ECUs is an optimization problem in itself [28], and the simulation environment uses a greedy approach to deal with this. A path between the sender and receiver of a signal is decided using the breadth first search algorithm (BFS) [29, 30], which finds a shortest path, in terms of the number of components the signal has to traverse.

Once a path has been assigned to each inter-ECU signal, the simulation environment determines how often each signal will be transmitted. As described in Section 2.2, each signal has an updating time U_s that specifies how often its value may be updated. For sporadic signals, the updating time corresponds to the minimum time separation between consecutive initiations, see Section 2.2, which means that these signals are treated in the same manner as periodic signals. The traversed hardware components might however not be able to broadcast a signal exactly when it is updated. The period time T_s^b describes how often signal s is broadcast on bus b, and its value depends on the dynamics of the bus protocol. The computations of T_s^b for CAN, LIN, and FlexRay are described in the three paragraphs below. For gateways and ECUs, the dynamics have been simplified by allowing for these hardware components to initiate the transmission of a signal exactly when the signal value has been updated.

CAN Since the CAN protocol is event-triggered, there is no period time associated with the bus itself [17]. The period time for transmission of a signal is thus equal to the updating time of that signal, i.e. $T_s^b = U_s$ [4].

LIN The period time T_s^b for signal s transmitted over LIN b can, with the help of the schedule tables described in Subsection 2.5.2, be adapted to the requirements on the signal. The model for period time calculation, developed in this work, considers the delivery time of a signal's transmission over the hardware network, R_s, and its E2E time D_s. The aim of the model is to ensure that all signals are received before their E2E times. How the delivery time is computed is described in Section 3.4.

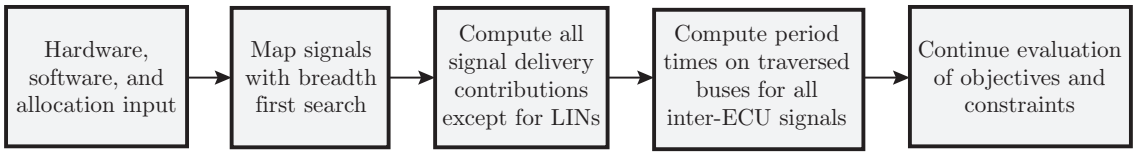


Figure 3.1: ***The simulation environment procedure.*** *Steps 2, 3, and 4 constitute the pre-evaluation procedure.*

Table 3.1: ***Overhead factors for CAN, LIN and FlexRay.*** *Each factor ρ_b^{oh} is obtained by dividing the size of the entire frame with the size of the payload, which contains the signal values. Note that ρ_b^{oh} for LIN is multiplied with* 1.4 *in order to account for the* 40% *increased time budget for each frame, described in Subsection 2.5.2. After [4].*

Bus	CAN	LIN	FlexRay
Overhead size	97 bit	44 bit	86 bit
Payload size	64 bit	64 bit	240 bit
Overhead factor ρ_b^{oh}	2.52	2.35	1.36

The delivery time contributions of all other hardware components, except for the LIN bus, are summed to R_s^{-b}, and the remaining available delivery time over the LIN bus is thereafter computed by subtracting R_s^{-b} from D_s. This difference is further reduced by subtracting τ_s^b, the transmission time over LIN for signal s. The reason for this is that the signal value, in the worst case, is updated just after the previous transmission has occurred, and the new signal value thus has to wait τ_s^b before transmission. The transmission time also imposes a lower boundary for the period time, since a new instance of the signal cannot be transmitted while the previous instance is still being broadcast. An upper bound for the period time is set by the updating time U_s of the signal, since longer period times may result in unsent signal values being overwritten and lost. In summary, the period time for LIN is given by

$$T_s^b = \begin{cases} \tau_s^b, & \text{if } D_s - R_s^{-b} - \tau_s^b \leq \tau_s^b \\ D_s - R_s^{-b} - \tau_s^b, & \text{if } \tau_s^b < D_s - R_s^{-b} - \tau_s^b \leq U_s \\ U_s, & \text{if } U_s < D_s - R_s^{-b} - \tau_s^b. \end{cases} \tag{3.1}$$

FlexRay The period time for a signal transmitted over FlexRay is determined according to the time length of a communication cycle for the FlexRay bus, T_{FR}. Signals with long updating times could be transmitted less frequently in order to decrease their busload contributions. This is however not considered for this model. The period time for signal s is set to $T_s^b = T_{\text{FR}} = 5\,\text{ms}$.

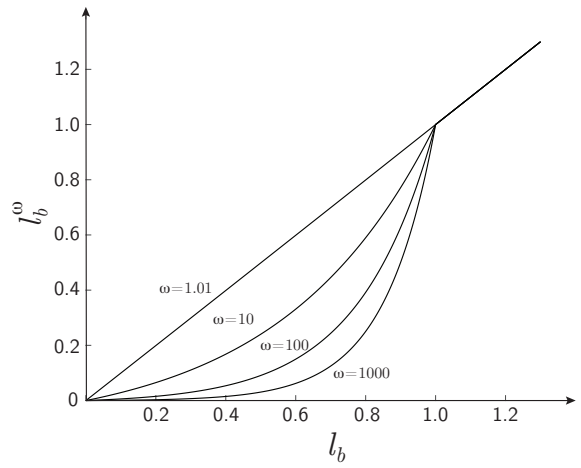
3.2 Busload

One extensive study which models the busload of different protocols was made by Hardung [4]. In his doctoral thesis, he suggests a model for estimating the busload of a single bus. The busload l_b of bus b is estimated by

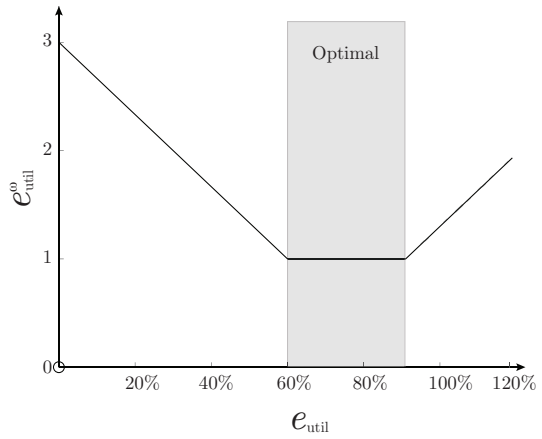
$$l_b = \sum_{s \in \boldsymbol{S}_b} \frac{\rho_b^{\text{oh}} L_s}{r_b T_s^b}, \tag{3.2}$$

where r_b is the bit rate of the bus, $\boldsymbol{S}_b$ is the set of signals that are transmitted over the bus, L_s is the size of signal s in bits, T_s^b is the signal's period time on bus b, and ρ_b^{oh} is the overhead factor, i.e. the ratio between the size of a whole frame and the size of the payload [4]. The overhead factors used for LIN, CAN, and FlexRay are presented in Table 3.1. These values are used assuming that signals can be packed optimally into each bus frame, meaning that no bits in the payload are unused [4].

The value l_b gives an unbiased measurement of how much of the bus' bandwidth is utilized. Hardung [4] argues that the model should disfavor very high busload, and has therefore introduced a weight function. The function can be written as



(a) ***The weight function for busload for different values of the weight parameter ω.*** *A higher weight implies a increased tolerance for high busloads.*



(b) ***The weight function for memory utilization.*** *The optimal memory utilization is between 60% and 90% of the available memory. Note that an ECU with $e_{\text{util}} = 0$ obtains $e_{\text{util}}^\omega = 0$.*

Figure 3.2: ***Weight functions used for busload and memory utilization.*** *Reproduced with permission [4, 8].*

$$l_b^\omega = \begin{cases} \frac{\omega^{l_b}-1}{\omega-1} & \text{if } l_b < 1 \\ l_b & \text{otherwise,} \end{cases} \tag{3.3}$$

where ω is the weight and l_b^ω is the busload after it has been weighted. A plot with the function for different values of ω is shown in Figure 3.2a. The weight $\omega = 100$ will be used throughout this work.

The objective value f_{busload} for an allocation $\boldsymbol{x}$ is calculated as the sum of l_b^ω for all buses

$$f_{\text{busload}}(\boldsymbol{x}) = \sum_{b \in \boldsymbol{B}} l_b^\omega, \tag{3.4}$$

where $\boldsymbol{B}$ is the set of buses [4]. In this work, the busload is also implemented as a constraint, since $l_b > 1$ is not feasible. The total constraint violation $g_{\text{busload}}(\boldsymbol{x})$ of $\boldsymbol{x}$ is calculated as

$$g_{\text{busload}}(\boldsymbol{x}) = \sum_{b \in \boldsymbol{B}} \max(l_b^\omega - 1, 0). \tag{3.5}$$

3.3 Memory utilization

The second objective considered in this work is the utilization of the ECU memories. How much of an ECU's memory that is currently occupied is determined by the sizes of the SWCs currently allocated to it, and is calculated as

$$e_{\text{util}} = \sum_{c \in \boldsymbol{C}_e} \frac{c_{\text{size}}}{e_{\text{cap}}}, \tag{3.6}$$

where e_{util} is the memory utilization of ECU e, c_{size} is the size of SWC c, e_{cap} is the space available on ECU e, and $\boldsymbol{C}_e$ is the set of SWCs allocated to ECU e.

In [8], Dohr & Eichberger argue that optimal memory utilization is somewhere in the interval of $[60\%, 90\%]$ of the available space of the ECU. Their weight function, presented in Figure 3.2b, is therefore used in this work. If the utilized memory of the ECU is in the mentioned interval, that ECU's memory utilization is weighted as $e_{\text{util}}^\omega = 1$. Otherwise, e_{util}^ω increases linearly with the distance from the interval. A special case

is made for empty ECUs. Since unused ECUs may be removed, resulting in a reduced cost for the hardware architecture, empty ECUs are weighted as $e^{\omega}_{\text{util}} = 0$. Evaluating all ECUs in the architecture this way enables the calculation of the objective value through

$$f_{\text{memory}}(\boldsymbol{x}) = \sum_{e \in \boldsymbol{E}} e^{\omega}_{\text{util}}, \tag{3.7}$$

where $\boldsymbol{E}$ is the set of ECUs in the hardware architecture.

It is worth noting that the weight function makes no distinction between allocations that are feasible, i.e. under 100% memory utilization, and infeasible ones, i.e. over 100%. This work thus also models memory utilization as a constraint, with the constraint violation being calculated as

$$g_{\text{memory}}(\boldsymbol{x}) = \sum_{e \in \boldsymbol{E}} \max(e_{\text{util}} - 1, 0) \tag{3.8}$$

3.4 Signal delivery times

As described in Section 2.2, many signals within the software architecture of a vehicle have requirements in regard to the delivery time from sender SWC to receiver SWC. In order to evaluate whether these constraints are satisfied, the simulation environment uses worst-case analysis. While this approach may overestimate the actual delivery times for signals, the ability to provide guarantees for safety-critical functionalities outweights this drawback [12].

In the case where the sender and receiver SWCs of a certain signal are allocated to the same ECU, the simulation environment approximates that this signal always experiences an acceptable delivery time. In the opposite case, the delivery time is compared to the E2E time of the signal. The constraint violation of an allocation $\boldsymbol{x}$ is given by

$$g_{\text{delivery}}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{S}|} \sum_{s \in \boldsymbol{S}} 1\{R_s > D_s\}, \tag{3.9}$$

where $\boldsymbol{S}$ is the set of signals in the software architecuture, $|\boldsymbol{S}|$ is the number of signals in that set, R_s is the worst-case delivery time of signal s, D_s is the signal's end-to-end time, and $1\{P\}$ denotes the function that returns 1 if P is true and 0 otherwise.

Each of the hardware components that an inter-ECU signal traverses during transmission contributes to the total delivery time R_s. Similarly to previous work [4, 9], this work simplifies the delivery time contributions from ECUs and gateways, by modeling these as 3 ms delays. The estimation of delivery times over buses are based on previously developed models [4, 12, 14]. All of these compute the worst-case delivery time R^b_s of a signal s on bus b as

$$R^b_s = \tau^b_s + w^b_s, \tag{3.10}$$

where τ^b_s denotes the transmission time on the bus, and w^b_s is the time that the signal may be required to wait before being transmitted over the bus [4]. The models used for CAN, LIN and FlexRay are presented in paragraphs below.

CAN As mentioned in Subsection 2.5.1, there exists a well-tested approach to estimate the worst-case delivery times on CAN [12, 14]. This analysis takes bandwidth, signal size and bit stuffing into account when computing the transmission time τ^b_s for a signal. In order to provide an upper bound for τ^b_s, it is assumed the maximum amount of stuffed bits are included in the frame and, as opposed to the busload computation, each CAN-frame only contains one signal.

To compute the waiting time w^b_s, the analysis assumes that signals are queued repeatedly with their respective updating times U_s. One signal is removed from the queue each time the bus is free, and the removal is performed according to the priority-based arbitration that CAN uses. The resulting delivery time is computed for each signal that is placed in the queue, which allows this approach to determine the worst-case delivery time for every signal on the CAN.

LIN The transmission time τ_s^b of a signal transmitted over LIN is computed using the model presented by Hardung [4]. This model considers the 40% increase in transmission time, mentioned in Subsection 2.5.2, in order to compute τ_s^b. As described in Section 3.1, the period time T_s^b for a signal transmitted over LIN is determined after all other delivery time contributions have been computed. Note that the first case in Equation (3.1) implies that the signal will not be received before its E2E time. Since a signal updated just after the previous transmission will have to wait an entire period before transmission, this means that the waiting time $w_s^b = T_s^b$. The worst-case delivery time over LIN is thus given by $R_s^b = \tau_s^b + T_s^b$.

FlexRay As noted in Subsection 2.5.3, a FlexRay cycle contains both static and dynamic slots. However, only the former is considered in the model. Every signal transmitted over FlexRay is thus assumed to have a constant period time $T_s^b = 5\,\text{ms}$ between consecutive transmissions, as described in Section 3.1. In order for a node to broadcast a signal during its slot, the signal value must be written to the bus before the start of the slot. This means that the worst-case waiting time w_s^b is equal to T_s^b, and it occurs in the case when a new signal value is received just after the node has initiated its broadcast.

To compute the transmission time τ_s^b over FlexRay, the model presented by Hardung [4] is used. This model provides a worst-case estimation by computing the time between the sender node transmitting the first bit in the slot and the receiver nodes obtaining the last bit in the slot.

3.5 Hardware demand

The simulation environment models the hardware demands mentioned in Section 1.2 by allowing an SWC to have a subset of the available ECUs as legal allocations [3]. $\boldsymbol{E}_c$ denotes the set of ECUs that constitutes legal allocations for SWC c. Thus, $\boldsymbol{E}_c = \boldsymbol{E}$ if an SWC does not have any specific hardware demands. The value of the constraint violation in regard to hardware demand is evaluated through

$$g_{\text{hardware}}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}|} \sum_{c \in \boldsymbol{C}} 1\{e_c \notin \boldsymbol{E}_c\} \tag{3.11}$$

where $\boldsymbol{C}$ is the entire set of SWCs, $|\boldsymbol{C}|$ is the number of SWCs in that set, e_c is the ECU that SWC c is allocated to, and $1\{P\}$ is the same function as in Equation (3.9).

3.6 Co-location requirement

The final constraint implemented in the simulation environment is the co-location requirements of SWCs. An SWC c that belongs to a co-location group has a set of SWCs, $\boldsymbol{C}_c^{\text{coloc}}$, that must be allocated to the same ECU as SWC c [3]. SWCs that do not belong to any co-location group have $\boldsymbol{C}_c^{\text{coloc}} = \emptyset$. The violation of the co-location constraint is evaluated through

$$g_{\text{colocation}}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}|} \sum_{c \in \boldsymbol{C}} 1\{\boldsymbol{C}_c^{\text{coloc}} \nsubseteq \boldsymbol{C}_{e_c}\}, \tag{3.12}$$

where $\boldsymbol{C}$ is the entire set of SWCs, $|\boldsymbol{C}|$ is the number of SWCs in that set, $\boldsymbol{C}_{e_c}$ is the set of SWCs allocated to ECU e_c, e_c is the ECU that SWC c is allocated to, and $1\{P\}$ is the same function as in Equation (3.9).

4
Optimization

This chapter presents the multi-objective genetic algorithm NSGA-II and the problem-specific genetic operators, together used for optimization of the software allocations in automotive architectures. Additionally, the first section gives brief introductions to multi-objective optimization, Pareto-dominance, and the concept of hypervolume.

4.1 Multi-objective optimization

A multi-objective optimization problem (MOOP) is, as the name indicates, a problem with more than one objective. Compared to single objective optimization problems, which often only have one global optimum, MOOPs can have multiple different optima that favor different objectives. A formal definition of an MOOP is

$$\text{minimize } \boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x}))^{\mathrm{T}} \tag{4.1}$$

$$\text{subject to } \boldsymbol{g}(\boldsymbol{x}) = (g_1(\boldsymbol{x}), g_2(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))^{\mathrm{T}} \leq (0, 0, \ldots, 0)^{\mathrm{T}}, \tag{4.2}$$

where $\boldsymbol{x} = (x_1, x_2, \ldots, x_q)^{\mathrm{T}}$ is a solution, $\boldsymbol{f}(\boldsymbol{x})$ is the set of m objectives and $\boldsymbol{g}(\boldsymbol{x})$ is the set of n constraints [4]. A solution $\boldsymbol{x}$ is referred to as feasible if it does not violate any of the n constraints. In the context of software allocation, a solution describes how the set of SWCs are allocated. Software allocation problems generally include tight constraints, which cause a high percentage of the solutions within the search space to be infeasible [4]. The following section describes Pareto-dominance and Pareto-optimality, two key concepts within MOOPs.

4.1.1 Pareto-optimality

With multiple objectives to optimize, it can be hard to determine which solution solves the optimization problem best. In order to compare solutions, one approach is to utilize Pareto-dominance [31]. For solution $\boldsymbol{x}_1$ to dominate solution $\boldsymbol{x}_2$, $\boldsymbol{x}_1$ cannot be worse than $\boldsymbol{x}_2$ in any objective and must be better than $\boldsymbol{x}_2$ in at least one objective. Assuming minimization, $\boldsymbol{x}_1$ dominates $\boldsymbol{x}_2$ if

$$\exists\, i \in \{1, 2, \ldots, m\} :\ f_i(\boldsymbol{x}_1) < f_i(\boldsymbol{x}_2), \tag{4.3}$$

and

$$f_j(\boldsymbol{x}_1) \leq f_j(\boldsymbol{x}_2) \quad \forall\, j = 1, \ldots, m. \tag{4.4}$$

A Pareto-optimal solution $\boldsymbol{x}$ of an MOOP is defined as a solution which is not Pareto-dominated by any other solution. The set of Pareto-optimal solutions for an MOOP is called the Pareto-front. By definition, all solutions not included in the Pareto-front are objectively worse than at least one of the solutions in the Pareto-front. The goal of multi-objective optimization is thus to find the Pareto-front of the problem, or approximate it as well as possible. Once the Pareto-front has been obtained, the selection of one single solution could be made by weighting the different objectives against each other.

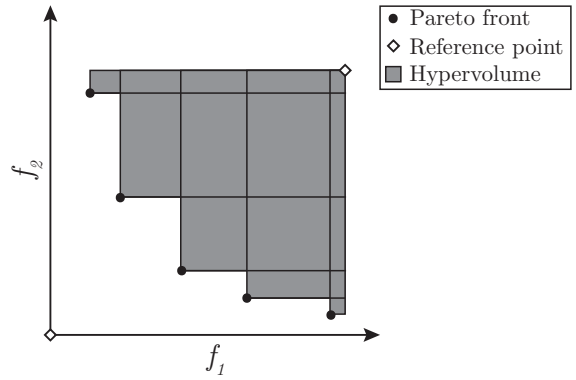


Figure 4.1: ***The hypervolume for the Pareto-front of a minimization problem with two objectives.*** *The value of the hypervolume is given by the area of the union of all rectangles, each formed by the reference point and one of the solutions within the Pareto-front.*

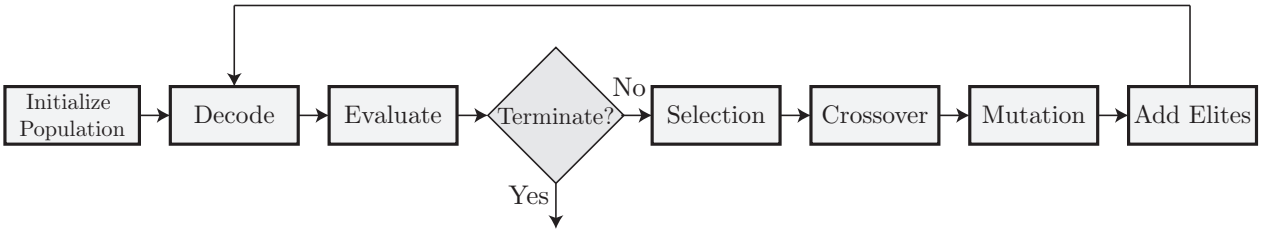


Figure 4.2: ***Flowchart of a standard genetic algorithm.*** *A new population is created each iteration by applying genetic operators on the previous generation. The algorithm is terminated if predefined criteria are met. This can for example be if a satisfactory fitness value has been achieved or a certain number of generations has passed.*

4.1.2 Hypervolume

One approach for evaluating a Pareto-front is to compute its hypervolume $\mathcal{H}$ [32]. The hypervolume of a front $\mathcal{F}$ is computed by first using each solution $\boldsymbol{x}_i \in \mathcal{F}$ and a reference point r to form hyperrectangles, by placing the two points in opposite diagonal corners. The value is then given by the hypervolume of the union of all such hyperrectangles. If the maximum possible value for each objective is known, $r = (f_1^{\max}, \ldots, f_m^{\max})$ is a useful reference point [33]. The hypervolume for the Pareto-front of a minimization problem with two objectives is illustrated in Figure 4.1.

If the true Pareto-front of a problem is known, its hypervolume can be compared to the hypervolume of an approximated Pareto-front. The hypervolume ratio $\mathcal{H}_{\text{ratio}} = \frac{\mathcal{H}_{\text{approx}}}{\mathcal{H}_{\text{true}}}$ is a useful metric for this comparison [34], and yields a value close to $\mathcal{H}_{\text{ratio}} = 1$ if the approximated Pareto-front corresponds well to the true Pareto-front.

4.2 Genetic algorithms

Genetic algorithms (GAs) are one of the most common methods for solving the problem of allocating software components [35]. Inspired by biology, and more specifically by genetics and the theory of evolution, Holland introduced this metaheuristic optimization method in 1975 [36]. The basic approach of a GA is to simulate natural selection and evolution in order to explore the search space of a problem and find high-quality solutions [37]. This is achieved by generating a set of candidate solutions, called individuals, to the optimization or search problem, and then subjecting these to operations that resemble evolutionary processes. The result is the that favorable genetic material is propagated to the individuals of future generations.

A flowchart of a general genetic algorithm is presented in Figure 4.2. Within the **initialization** step, the individuals are created by encoding the possible solutions of the optimization problem into strings (chromosomes) of values that, in turn, are referred to as genes [38]. Together, the created individuals form the initial

population, which is also the first generation. During each generation, the chromosome of every individual in the population is **decoded** back into its corresponding solution to the problem, a process that may require complex interactions between several genes [38]. The individual is then **evaluated** with regard to the objectives of the problem. The individual is assigned a fitness value according to its performance, as given by the fitness measure. By convention, a higher fitness should correspond to a better solution to the problem. The next step in the algorithm is to modify the population in order to create new solutions in the following generation. This procedure consists of a combination of genetic operators. The three main operators are selection, crossover, and mutation, but some versions also use elitism [38]. These four operators are briefly described in the following paragraphs.

Selection In the selection phase, individuals are chosen for procreation. In order to find better solutions, it is essential to choose individuals with high fitness values. However, always picking the best solutions creates a high selection pressure, which decreases the possibility for exploration of the search space and may result in premature convergence, i.e. the population converging towards a local optimum rather than a global one [37]. Selection is therefore generally performed stochastically [38], which allows individuals with low fitness values to breed as well.

Crossover Like selection, crossover is a convergence operator. By imitating nature's way of mixing the genetic material of two individuals, crossover allows a genetic algorithm to search for the most favorable combinations of the genetic material in the population [38].

Mutation In order to increase the diversity of genetic material in the population, a mutation operator is included that randomly changes the values of genes [38]. A mutation often leads to divergence from local optima. Despite this effect, the inclusion of mutation is often essential in the process of localization of global optima. The mutation operator decreases the risk of premature convergence.

Elitism The elitism operator is used to ensure that the current best combination of genetic material stays in the population. Since the use of the other operators can deform high fitness individuals, a number of copies of the hitherto best individual is kept in storage. When the generation of a new set of individuals is finished, the stored individuals replace a corresponding number of individuals in the new population.

This chain of processes results in a new set of individuals, which replace the previous generation. These individuals are ready to be decoded, evaluated and eventually modified by repeating the procedure [38]. The generational replacement continues until a satisfactory result has been reached, or a certain number of generations has passed.

4.2.1 Non-dominated sorting genetic algorithm II

A successful approach to optimization of software allocation has been to treat the problem as an MOOP and use a sub-category of GAs called multi-objective genetic algorithms (MOGAs) for the optimization [3, 4, 8]. MOGAs are specifically adapted to handling the problem of comparing solutions when there are more than one objective to take into account. Furthermore, the algorithms are designed to keep the population diverse and maintain a wide and evenly distributed set of non-dominated solutions [39].

A well-known MOGA, and the one used for this thesis, is the Non-dominated sorting genetic algorithm II (NSGA-II), developed by Deb et al. [31]. This is an improvement of the previous algorithm, NSGA [40], even though the algorithms bear little resemblance to each other except for the name. The purpose of improving the algorithm was to speed up the computation time, introduce elitism, and remove unnecessary parameters for increasing diversity by instead adding a parameterless diversity mechanism. Comparisons with another popular MOGA called Strength pareto evolutionary algorithm 2 (SPEA2) [41], show that NSGA-II works better for two objectives, but is outperformed by SPEA2 when the number of objectives is increased [4]. Moreover, NSGA-II seems to often present a wider spectrum of non-dominated solutions, while SPEA2 has less clustering within the set of non-dominated solutions and therefore a more even distribution [39]. The complete NSGA-II algorithm can be seen in Algorithm 4.1.

The most important aspect of NSGA-II is the introduction of an archive. This is an extension of the elitism concept, used in order to increase the performance of the algorithm and to prevent the loss of good solutions

Algorithm 4.1: NSGA-II

1:	Set $g = 1$	Initialize generation counter
2:	`initialize`(P_g) and set $O_g = \emptyset$	Initialize parent and child populations
3:	**while**	Repeat if stop criteria not met
4:	$R_g = P_g \cup O_g$	Combine parent and child population
5:	$\boldsymbol{\mathcal{F}} =$ `fast-non-dominated-sort`(R_g)	Sort R_g into non-dominated fronts $\boldsymbol{\mathcal{F}} = (\mathcal{F}_1, \mathcal{F}_2, \dots)$
6:	Set $P_{g+1} = \emptyset$ and $i = 1$	
7:	**while** $\lvert P_{g+1}\rvert + \lvert\mathcal{F}_i\rvert \leq \lvert P_g\rvert$ **do**	Add as many complete fronts as possible to the parent population
8:	`crowding-distance-assignment`$(\mathcal{F}_i)$	Calculate crowding distances within $\mathcal{F}_i$
9:	$P_{g+1} = P_{g+1} \cup \mathcal{F}_i$	Include ith non-dominated front in the parent population
10:	$i = i + 1$	Check the next front for inclusion
11:	Sort $\mathcal{F}_i$ according to $\mathcal{D}$	Sort in descending order of crowding distance
12:	$P_{g+1} = P_{g+1} \cup \mathcal{F}_i[1 : (\lvert P_g\rvert - \lvert P_{g+1}\rvert)]$	Fill up the parent population
13:	$O_{g+1} =$ `make-new-pop`(P_{g+1})	Use genetic operators to generate child population O_{g+1}
14:	$g = g + 1$	Increment the generation counter

Algorithm 4.1: *The steps of the algorithm are explained further in Subsection 4.2.1.*

[31]. The archive, or parent population, is denoted by P in Algorithm 4.1, and is used when creating the new child population O. Lines 4 through 12 of Algorithm 4.1 describe the process of updating the parent population during each generation.

Each new generation starts by combining the parent and child populations from the previous generation, creating a combined population R. This is then used to form the new parent population by taking the most beneficial genetic material. The process of determining which genetic material is most beneficial is incorporated in NSGA-II. The individuals are ranked using a method which starts by sorting the individuals of the combined population into fronts. This is done with *fast non-dominated sort*, also developed by Deb et al. [31]. A front $\mathcal{F}_i$ is a set of solutions that are not dominated by any of the other solutions in that front. Front $\mathcal{F}_1$ is therefore the current set of non-dominated solutions in the combined population. In order to obtain $\mathcal{F}_2$, the solutions in $\mathcal{F}_1$ are temporarily removed and the new set of non-dominated solutions is selected.

The NSGA-II algorithm uses an extension of the definition of Pareto-domination, presented in Subsection 4.1.1, which includes the possibility of infeasible solutions. The purpose of this extension is to never weight objective values against constraint violations. If two feasible solution are compared, the original domination definition is used. With one feasible and one infeasible solution, the feasible solution always dominates the infeasible one, regardless of their objective values. In the case that both solutions are infeasible, their dominance relation is established only through their overall constraint violations $G(\boldsymbol{x})$ [42]. Given the set of constraints $\boldsymbol{g}(\boldsymbol{x})$ for a solution $\boldsymbol{x}$, the overall constraint violation is computed as

$$G(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} \omega_i g_i(\boldsymbol{x})}{\sum_{i=1}^{n} \omega_i} \tag{4.5}$$

where $\omega = 1/g_i^{\max}$ is a weight parameter for constraint i, $g_i^{\max}$ is the maximum constraint violation so far for constraint i and n is the number of constraints [42]. This means that the weight parameters $\boldsymbol{\omega} = \{\omega_1, \dots, \omega_n\}$ may vary between generations. The reason behind including weight parameters is to balance the contribution from each constraint to the overall constraint violation, and their respective values are updated during the evolution since the maximum constraint violation might not be known beforehand.

The last step before forming the new parent population is to rank the individuals within each front. This ranking is made with regard to objective density, in order to promote diversity in the population. The measurement for density is called crowding distance $\mathcal{D}$ and its computation is presented in Algorithm 4.2. The individuals that have the edge values for each objective are given $\mathcal{D} = \infty$, which means that they receive the highest ranks within the front. The remaining individuals are given crowding distances according to their nearest neighbor distances for each objective. This is scaled by the highest and lowest objective values obtained so far, $f_j^{\max}$ and $f_j^{\min}$. A visualization of this computation is provided in Figure 4.3, where the

Algorithm 4.2: Crowding distance assignment

Input: Front $\mathcal{F}$
Output: Crowding distance $\mathcal{D}_i$ for each individual in $\mathcal{F}$

1:	**for** $i = 1 : \lvert\mathcal{F}\rvert$ **do**	Loop over all individuals in $\mathcal{F}$
2:	$\mathcal{D}_i = 0$	Initialize crowding distances
3:	**for all** objective f_j **do**	
4:	Sort $\mathcal{F}$ according to f_j	Sort in ascending order of current objective
5:	Sort $\mathcal{D}$ in same order as $\mathcal{F}$	$\mathcal{D}_i$ should correspond to individual i in $\mathcal{F}$
6:	$\mathcal{D}_1 = \mathcal{D}_{\lvert\mathcal{F}\rvert} = \infty$	Make sure that edge points get selected
7:	**for** $k = 2 : \lvert\mathcal{F}\rvert - 1$ **do**	For all other individuals
8:	$\mathcal{D}_k \mathrel{+}= \frac{f_j(k+1) - f_j(k-1)}{f_j^{\max} - f_j^{\min}}$	Compute nearest neighbor distances for current objective and normalize with current range for the objective

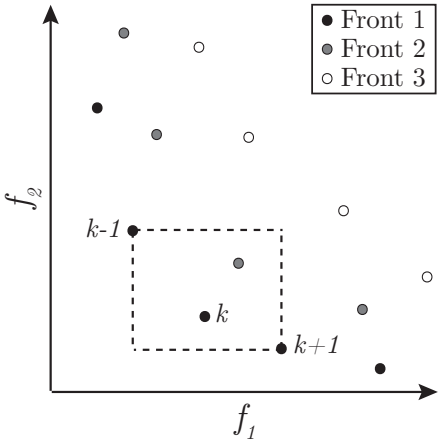


Figure 4.3: ***Crowding distance calculation with two objectives.*** *The crowding distance of individuals k is the average side length of the dashed rectangle, which is obtained by computing the distances to the objective values of two nearest neighbours within the same front. Note that the normalization factor for each objective is neglected in this figure.*

individuals of the current front are marked by black dots and the crowding distance of individual k is the average side length of the dashed rectangle.

By sorting each front in order of descending crowding distance, every individual in the population may essentially be given a distinct rank $\mathcal{R} \in \{1, \ldots, |R|\}$, where $|R|$ is the number of individuals in the combined population. All individuals with rank $\mathcal{R} \leq |P|$ are used to form then the new parent population. A generation in NSGA-II ends by creating a new child population O from the updated parent population. The following section describes the details of the genetic operators used within `make-new-pop()`, specified on line 10 in Algorithm 4.1, as well the schemes used for encoding and initialization. These aspects are not determined by NSGA-II and thus have to be chosen according to the problem specification.

4.2.2 Problem-specific encoding, initialization, and genetic operators

Encoding the allocation of SWCs onto ECUs can be made in several different ways. The most common method is an SWC-based encoding scheme, where each gene of the chromosome represents one SWC and the value of each gene represents an ECU on which the SWC is allocated [3, 4, 8]. Other encoding schemes have been tested, such as ECU-based encoding, where each gene represents an ECU and contains a list of SWCs which are allocated to it [3]. For the implementation described in this report, the first alternative will be used since it is more tested in previous work and the following genetic operators have more intuitive implementations.

A schematic overview of the allocation encoding can be seen in Figure 4.4. In a solution $\boldsymbol{x} = (x_1, x_2, \ldots, x_q)^{\mathrm{T}}$, each index i corresponds to one SWC and the value of the variable x_i corresponds to the ECU that SWC i is

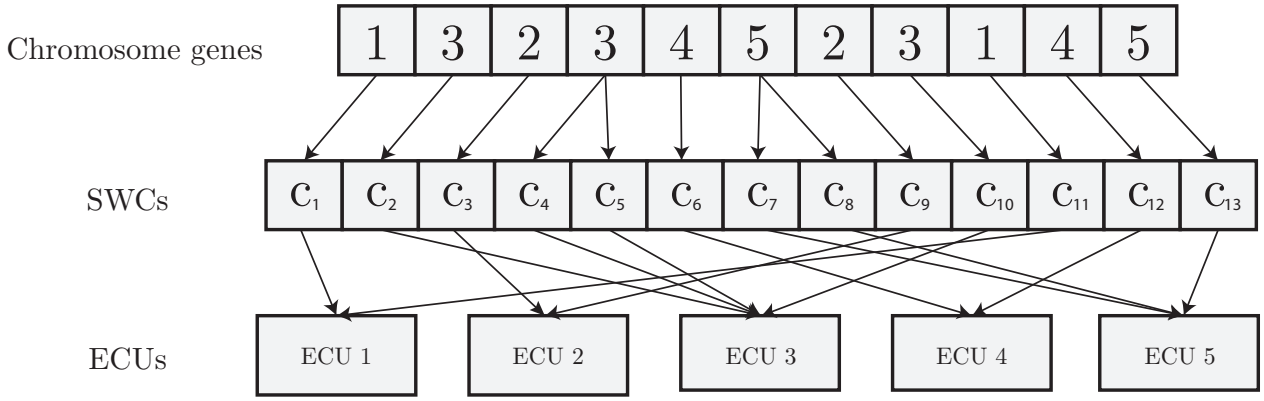


Figure 4.4: ***Encoding scheme used for the allocation of SWCs to ECUs.*** *A chromosome's gene corresponds to either an SWC or a co-location group of SWCs, and the value of the gene corresponds to the ECU allocation.*

allocated to. As seen in the figure, the possible values of x_i are integers, and each integer uniquely represents one ECU. In order to decrease the search space, groups of SWCs that demand co-location are encoded by a single gene in the chromosome. Apart from these special cases, each variable x_i in a solution is represented by one gene in the chromosome.

The initial parent population is generated by randomly assigning a value to each gene in every individual's chromosome. The random values are chosen from the set $1, 2, \ldots, |\boldsymbol{E}|$, where $|\boldsymbol{E}|$ is the number of ECUs.

For the selection process, each individual is given a fitness value F equal to its rank $\mathcal{R}$, described in Subsection 4.2.1. However, since this fitness assignment yields fitness values that are not proportional to the individuals' objective values, a fitness proportionate selection operator is unsuitable. Tournament selection on the other hand, which ignores the difference in fitness value between the compared individuals, is well adapted to the used fitness assignment [38]. The complete algorithm of this selection operator is presented in Algorithm 4.3. By randomly picking n_{tour} contestants from the parent population, sorting these according to fitness and performing a round-based tournament, one single individual gets selected for procreation. The selection pressure can be adjusted by varying the tournament selection parameter p_{tour}. Selection with replacement is used within the described implementation, which means that an individual may be selected for procreation multiple times.

Algorithm 4.3: Tournament selection

Input: Population P
Number of contestants n_{tour}
Tournament selection parameter p_{tour}
Output: One selected individual

1: Pick n_{tour} contestants randomly from P
2: Sort contestants according to fitness F — Sort in descending order
3: **for** $i = 1 : (n_{\text{tour}} - 1)$ **do** — Loop through individuals
4: $r =$ random $\in [0, 1)$
5: **if** $r < p_{\text{tour}}$ **then**
6: Return individual i
7: **else if** $i = n_{\text{tour}} - 1$ **then** — Only individual $i + 1$ left in the tournament
8: Return individual $i + 1$

Based on the chosen encoding scheme, there are a number of possible crossover operators. For this work, the two-point crossover is used. The full algorithm is presented in Algorithm 4.4 together with a clarifying image in Figure 4.5. By choosing two random points in the interval $[1, m]$, where m is the length of the chromosome, and then cutting the two chromosomes at these points, subparts of chromosomes are created. By combining these subparts according to the figure, two new individuals are created. An important note is that even

Algorithm 4.4: Two-point crossover

Input: Parent chromosomes $\mathcal{C}_1^p$, $\mathcal{C}_2^p$ with length m
Output: Child chromosomes $\mathcal{C}_1^o$, $\mathcal{C}_2^o$

Set $\mathcal{C}_1^o = \mathcal{C}_2^o = \emptyset$
r_1 = random integer $\in [1, m]$
r_2 = random integer $\in [1, m]$
$i_1^{\text{cut}} = \min(r_1, r_2)$
$i_2^{\text{cut}} = \max(r_1, r_2)$
for $j = 1 : m$ **do**
    if $j < i_1^{\text{cut}}$ **then**
        Add gene j from $\mathcal{C}_1^p$ to $\mathcal{C}_1^o$
        Add gene j from $\mathcal{C}_2^p$ to $\mathcal{C}_2^o$
    else if $i_1^{\text{cut}} \leq j < i_2^{\text{cut}}$ **then**
        Add gene j from $\mathcal{C}_2^p$ to $\mathcal{C}_1^o$
        Add gene j from $\mathcal{C}_1^p$ to $\mathcal{C}_2^o$
    else
        Add gene j from $\mathcal{C}_1^p$ to $\mathcal{C}_1^o$
        Add gene j from $\mathcal{C}_2^p$ to $\mathcal{C}_2^o$

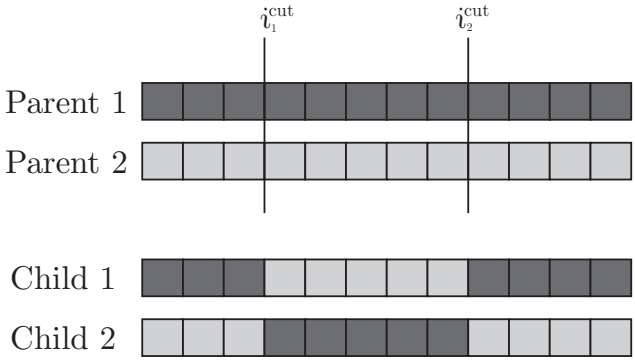


Figure 4.5: ***Two-point crossover.*** *Each parent chromosome is split at two random points and the subparts are then combined to form the child chromosomes.*

Algorithm 4.5: Swap mutation

Input: Chromosome $\mathcal{C}$ with length m,
Swap mutation probability p_{swap}
Output: Mutated chromosome

for $i = 1 : m$ **do**
    r_1 = random $\in [0, 1)$
    if $r_1 < p_{\text{swap}}$ **then**
        r_2 = random integer $\in [1, m]$
        Swap value of gene i in chromosome $\mathcal{C}$ with value of gene r_2 in chromosome $\mathcal{C}$

Algorithm 4.6: Point mutation

Input: Chromosome $\mathcal{C}$ with length m,
Point mutation probability p_{point}
Set of possible values $\mathcal{V}$
Output: Mutated chromosome

for $i = 1 : m$ **do**
    r_1 = random $\in [0, 1)$
    if $r_1 < p_{\text{point}}$ **then**
        r_2 = random value $\in \mathcal{V}$
        Change value of gene i in chromosome $\mathcal{C}$ to r_2

though a crossover is performed with chromosomes that encode feasible solutions, there is no guarantee that the resulting, mixed chromosomes will encode feasible solutions. However, two-point crossover ensures that a generated chromosome allocates each SWC to exactly one ECU. This can be guaranteed because of the used encoding scheme and the fact that the swapped subparts represent the same SWCs.

Two mutation operators are used to increase the diversity of the genome. These are called point and swap mutation and, similarly to the two-point crossover operator, they are adapted to the chromosome encoding in order to ensure that each SWC only allocates to one ECU [3]. The swap mutation algorithm is presented in Algorithm 4.5, and Algorithm 4.6 shows the point mutation algorithm. Both mutation operators check each gene in the chromosome and mutate it with the probabilities p_{swap} and p_{point} respectively. If a gene is subjected to swap mutation, the operator randomly chooses another gene in the chromosome and swaps the values of the two genes. A point mutated gene is assigned a random value from a set of possible values. The mutation operators were implemented in such a way that each gene in a chromosome is first checked for point mutation, and thereafter every gene is checked for swap mutation. This implementation of the mutation operator may result in an infeasible solution being generated, but the exploration of infeasible regions of the search space may be of importance within multi-objective optimization [43].

The optimization of software allocation does not have any inherent target objective values, i.e. busload and memory utilization, that the genetic algorithm should strive to obtain. For this reason, the termination criterion for the algorithm is set to a certain number of generations G. This concludes the description of the optimization and the following chapter presents the process of evaluating the validity of the method.

5
Evaluation process

This chapter provides a description of the developed framework for evaluation of the optimization method. The framework produces data resembling real automotive architectures, and the first section provides the details of the generation. This includes how parameter values are set and how signals and buses are routed in the software and hardware architectures respectively. Two different scenarios, each containing a certain parameter setup for the generation framework, are presented in the second section. This section also describes how optimizations were evaluated using the architectures generated according to these scenarios.

5.1 Framework for generation of scenarios

The amount of detailed automotive architecture data provided by car manufacturers may be limited [44], as in the case of this work. Therefore, a common approach, used by previous studies [3, 8, 9], has been to evaluate optimization methods using artificial data that resembles real automotive architectures. Generating data this way has both advantages and disadvantages. Compared to an industrial case study, which contains a limited amount of hardware and software to base the evaluation on, there are only practical restrictions to the amount of data that may be generated [44]. Generation of automotive architectures thus allows for a more thorough investigation of the capabilities and limitations of an optimization method. On the other hand, it can be difficult to generate automotive architectures that fully capture the essence of real architectures, making the validity of the model for real-world applications hard to confirm. Additionally, it is hard to generate automotive architectures where the optimization of software allocation is tightly constrained, while at the same time ensuring that feasible solutions do in fact exist. Inspired by the previous studies, a method for generating automotive architecture data has been developed.

5.1.1 Component parameters

The generation of an automotive architecture begins by creating all of its components, and setting parameter values for each of these. A parameter value is chosen in one of four different ways. It can either be set by the user, set relatively to other parameters, selected randomly from a predefined range, or be static, i.e. fixed and independent of other parameters. A full parameter description, except for the input parameters, can be seen in Table 5.1. The input parameters are the number of SWCs $|\boldsymbol{C}|$, the number of CAN buses $|\boldsymbol{B}_{\text{CAN}}|$, the number of FlexRay $|\boldsymbol{B}_{\text{Flex}}|$, the number of ECUs $|\boldsymbol{E}|$, and the number of domains $|\boldsymbol{D}|$. See Section 2.2 for a description of domains.

ECUs have one parameter, the memory capacity e_{cap}. The value is generated randomly, with a uniform probability distribution, in the range of $[256\,\text{byte}, 1024\,\text{byte}]$ for each ECU. The sum of e_{cap} for all ECUs, denoted $\boldsymbol{E}_{\text{cap}}$, is then used as reference for assigning the memory consumption of the SWCs, c_{size}. This is done by choosing an integer from the interval $\left[\frac{\boldsymbol{E}_{\text{cap}}}{2|\boldsymbol{C}|}, \frac{\boldsymbol{E}_{\text{cap}}}{|\boldsymbol{C}|}\right]$ [3]. These assignments imply that approximately 75% of the total available ECU memory will be occupied by the SWCs and therefore yield relatively tight constraints on the allocation. Furthermore, the set of SWCs are uniformly distributed over the set of domains $\boldsymbol{D}$. 40% of the SWCs belong to a co-location group, and each such group consists of $|\boldsymbol{C}_c^{\text{coloc}}| = 3$ SWCs. The SWCs within a co-location group must belong to the same domain. Hardware demands are assigned to 10% of the set of SWCs, though an SWC cannot both belong to a co-location group and have hardware constraints. For each SWC with hardware demands, a third of the available ECUs are randomly chosen to be acceptable allocations.

Table 5.1: ***The parameters used for the generation of automotive architectures.*** *randi(min,max) generates a random integer between min and max and rand(**S**) chooses a random value from the set **S**.*

Parameter	Value	Comment
	ECU	
Memory capacity e_{cap} [kB]	randi(256,1024)	The total available ECU memory $\boldsymbol{E}_{\text{cap}} = \sum_{e \in \boldsymbol{E}} e_{\text{cap}}$
	SWC	
Memory consumption c_{size} [kB]	randi($\frac{\boldsymbol{E}_{\text{cap}}}{2\lvert\boldsymbol{C}\rvert}$, $\frac{\boldsymbol{E}_{\text{cap}}}{\lvert\boldsymbol{C}\rvert}$)	The sum of all SWC sizes is thus close to 75% of the total ECU memory.
Domain d	$d \in \{1, \ldots, \lvert\boldsymbol{D}\rvert\}$	SWCs are distributed uniformly over the set of domains.
Co-location group size $\lvert\boldsymbol{C}_c^{\text{coloc}}\rvert$	3	
Number of co-location groups	$0.4 \cdot \frac{\lvert\boldsymbol{C}\rvert}{\lvert\boldsymbol{C}_c^{\text{coloc}}\rvert}$	40% of all SWCs have co-location requirements.
Number of legal ECU allocations $\lvert\boldsymbol{E}_c\rvert$	$\frac{\lvert\boldsymbol{E}\rvert}{3}$	Only applicable if SWC has hardware demands
Number of SWCs with hardware demands	$0.1 \cdot \lvert\boldsymbol{C}\rvert$	10% of all SWCs have hardware demands.
	Vehicle feature	
Features per SWC factor r_c	$\frac{1.5}{\lvert\boldsymbol{C}\rvert_F^{\min}}$	See Subsection 5.1.1.
Number of SWCs in a feature $\lvert\boldsymbol{C}\rvert_F$	randi($\lvert\boldsymbol{C}\rvert_F^{\min}, \lvert\boldsymbol{C}\rvert_F^{\max}$)	$\lvert\boldsymbol{C}\rvert_F^{\min} = 4$, $\lvert\boldsymbol{C}\rvert_F^{\max} = 6$.
Number of sensors in a feature $\lvert\boldsymbol{K}\rvert_F$	randi($\lvert\boldsymbol{K}\rvert_F^{\min}, \lvert\boldsymbol{K}\rvert_F^{\max}$)	$\lvert\boldsymbol{K}\rvert_F^{\min} = 2$, $\lvert\boldsymbol{K}\rvert_F^{\max} = 3$.
Number of actuators in a feature $\lvert\boldsymbol{A}\rvert_F$	randi($\lvert\boldsymbol{A}\rvert_F^{\min}, \lvert\boldsymbol{A}\rvert_F^{\max}$)	$\lvert\boldsymbol{A}\rvert_F^{\min} = 2$, $\lvert\boldsymbol{A}\rvert_F^{\max} = 3$.
Number of features $\lvert\boldsymbol{F}\rvert$	$r_c \cdot \lvert\boldsymbol{C}\rvert$	See Subsection 5.1.1.
Domain d	$d \in \{1, \ldots, \lvert\boldsymbol{D}\rvert\}$	Features are distributed uniformly over the set of domains.
	Sensor	
Number of sensors $\lvert\boldsymbol{K}\rvert$	$\lvert\boldsymbol{F}\rvert$	See Subsection 5.1.1.
Domain d	$d \in \{1, \ldots, \lvert\boldsymbol{D}\rvert\}$	Sensors are distributed uniformly over the set of domains.
	Actuator	
Number of actuators $\lvert\boldsymbol{A}\rvert$	$\lvert\boldsymbol{F}\rvert$	See Subsection 5.1.1.
Domain d	$d \in \{1, \ldots, \lvert\boldsymbol{D}\rvert\}$	Actuators are distributed uniformly over the set of domains.
	Signal	
Number of signal senders $\lvert\boldsymbol{S}_{\text{send}}\rvert$	$\lvert\boldsymbol{C}\rvert + \lvert\boldsymbol{K}\rvert$	One transmitted signal may have several receivers.
Size L_s [byte]	randi(1,4)	
E2E time D_s [s]	rand({ 0.02, …, 0.2 })	In steps of 5 ms.
Updating time U_s [s]	D_s	Set to same value as E2E time.
Jitter time J_s [s]	0.001	Release jitter for scheduling on CAN [12, 14].
Priority P_s	$P_s \in \{1, \ldots, \lvert\boldsymbol{S}_{\text{send}}\rvert\}$	Assigned in order of decreasing E2E time.
	Gateway	
Number of gateways	1	See Subsection 5.1.1.
	LIN	
Number of sensors or actuators connected to LIN $\boldsymbol{H}_{\text{LIN}}$	2	Connected hardware components must belong to the same domain.
Number of LIN buses $\lvert\boldsymbol{B}_{\text{LIN}}\rvert$	$\frac{1}{2} \cdot \frac{\lvert\boldsymbol{K}\rvert + \lvert\boldsymbol{A}\rvert}{\boldsymbol{H}_{\text{LIN}}}$	See Subsection 5.1.1.

The connections, or signal routing, between SWCs, sensors, and actuators are made by forming features, similar to the one shown in Figure 2.3. This method is an extension of the one presented by [9], with the addition of each feature belonging to a specific domain. The number of features $|\boldsymbol{F}|$ generated depends on $|\boldsymbol{C}|$ and the static parameter r_c, which is called the features per SWC factor and affects how many features an SWC belong to on average. r_c in turn depends on another static parameter, namely the minimum number of SWCs in a feature $|\boldsymbol{C}|_F^{\min}$, through the relationship $r_c = \frac{1.5}{|\boldsymbol{C}|_F^{\min}}$. Setting $|\boldsymbol{F}| = r_c \cdot |\boldsymbol{C}|$ results in the average number of features that each SWC belongs to, to be above 1.5. To yield some variety in the generated features, the number of SWCs per feature, $|\boldsymbol{C}|_F$, is chosen randomly from the range $\left[|\boldsymbol{C}|_F^{\min}, |\boldsymbol{C}|_F^{\max}\right]$, where the limits are static parameters.

The number of sensors and the number of actuators that are included in a feature, $|\boldsymbol{K}|_F$ and $|\boldsymbol{A}|_F$, are selected randomly in the ranges of $[|\boldsymbol{K}|_F^{\min}, |\boldsymbol{K}|_F^{\max}]$ and $[|\boldsymbol{A}|_F^{\min}, |\boldsymbol{A}|_F^{\max}]$. The numbers of sensors and actuators in the entire architecture, $|\boldsymbol{K}|$ and $|\boldsymbol{A}|$, are both set to be the same as $|\boldsymbol{F}|$. Compared to SWCs, sensors and actuators are more frequently reused in different features [9], and the used values imply that each sensor and actuator is used in 2.5 features on average.

The generated software architecture contains $|\boldsymbol{S}_{\text{send}}| = |\boldsymbol{C}| + |\boldsymbol{K}|$ unique transmitted signals, or signal groups, with each signal originating from a sensor or an SWC. The number of receivers per signal is determined through the feature generation described above, and is thus not known during this stage of the process. The size of a signal, L_s, is a randomly chosen integer within the range $[1\,\text{byte}, 4\,\text{byte}]$ and its E2E time D_s is randomly chosen from $\{0.02\,\text{s}, 0.025\,\text{s}, \ldots, 0.2\,\text{s}\}$. As done in [9], a simplification has been made by setting the updating time of a signal, U_s, equal to its E2E time. The jitter time of a signal, which represents the longest time it may take between the initiation of a signal transmission and the signal being queued on a CAN bus [12, 14], is set to 0.001 s for all signals. Signal priorities, used for arbitration when transmitting on CAN, are assigned in order of decreasing E2E time, i.e. the signal with the lowest E2E time has the highest priority.

The hardware architecture consists of subnetworks, with the number of subnetworks being equal to $|\boldsymbol{B}_{\text{CAN}}| + |\boldsymbol{B}_{\text{Flex}}|$. ECUs, sensors, and actuators are connected to each subnetwork, resembling the connections illustrated in Figure 2.2. The set of subnetworks are joined using a gateway. Half of the sensors and half of the actuators within the architecture are connected to the subnetworks via ECUs, using either LIN or CAN. These connections are always made with groups of two sensors or two actuators, and the connected components must belong to the same domain. The presented framework makes these connections using LIN buses, provided that the requirements on the transmitted signals allow for it, see Subsection 5.1.3. For this reason, the number of LIN buses, $|\boldsymbol{B}_{\text{LIN}}| = \frac{1}{2} \cdot \frac{|\boldsymbol{K}|+|\boldsymbol{A}|}{\boldsymbol{H}_{\text{LIN}}}$, and the total number of CAN buses are only preliminary.

With all components generated, the next step is to form connections between these. The following sections describe how to connect the SWCs with sensors and actuators to form functionalities, and how to connect the hardware components to form the hardware network.

5.1.2 Signal routing in vehicle features

Each feature is generated by first choosing a number of SWCs, sensors, and actuators according to the parameters $|\boldsymbol{C}|_F$, $|\boldsymbol{K}|_F$, and $|\boldsymbol{A}|_F$. These components are generally taken from the pool of currently unused components belonging to the same domain as the feature, in order to ensure that all components are used in at least one feature. However, to model that some features include components from another domain, there is a probability $p = 0.01$ that such a component is chosen instead. If all SWCs, sensors, or actuators in a domain have been used in at least one feature and the corresponding pool of unused components is empty, already used components are reused.

The signal routing within a feature is made by connecting the chosen sensors with the actuators via the SWCs. This process is presented in Algorithm 5.1 and starts with placing the sensors in a queue called $Q_{\text{connected}}$. The next step is to place a subset of the SWCs in another queue called Q_{waiting}, which contains the components currently waiting to be connected to. How many SWCs to put in Q_{waiting} is decided by the layer size N_{size} parameter, and is set to 3 in this work. When this is done, the sensors in $Q_{\text{connected}}$ may connect to the components in Q_{waiting}, with a probability $p_{\text{connect}} = \frac{1}{3}$ for each possible connection. All

sensors in $Q_{\text{connected}}$ that were connected to at least one of the SWCs in Q_{waiting} are removed from the queue, and the SWCs in Q_{waiting} that were connected to are moved to $Q_{\text{connected}}$. The process is then repeated, by first filling up Q_{waiting} with SWCs until it contains N_{size} components again. If all SWCs already have been added to the Q_{waiting}, actuators are added instead. Actuators are however not moved from the Q_{waiting} if they have been connected to. When connecting the components in $Q_{\text{connected}}$ and Q_{waiting}, exceptions are made for connections that creates cycles. These are avoided by checking whether a path of signals from the SWC in the Q_{waiting} to the component in the $Q_{\text{connected}}$ has not already been created during the generation of a previous feature.

Algorithm 5.1: Connecting components in feature

Input: List of SWCs chosen for feature C
List of sensors chosen for feature K
List of actuators chosen for feature A
Layer size N_{size}
Connection Probability p_{connect}
Output: Feature

```
1:  Create empty lists Q_connected,Q_waiting
2:  Move all sensors from K to Q_connected
3:  while Q_connected not empty do
4:      while |Q_waiting| < N_size do
5:          if |C| > 0 then
6:              Move SWC from C to Q_waiting
7:          else
8:              Move actuator from A to Q_waiting
9:      for all q_connected ∈ Q_connected do
10:         for all q_waiting ∈ Q_waiting do
11:             if q_waiting does not have previous signal path to q_connected & rand(0,1) < p_connect then
12:                 Connect q_connected to q_waiting
13:                 if q_waiting is an SWC then
14:                     Mark q_waiting to be moved to Q_connected
15:         if q_connected was connected to any q_waiting ∈ Q_waiting then
16:             Remove q_connected from Q_connected
17:     Move all marked q_waiting ∈ Q_waiting to Q_connected
```

Even though the algorithm effectively generates features similar to real ones used in automotive architectures, it does not work every time. Due to the probabilistic choice of components and avoidance of cycles, the algorithm may not find a valid way of connecting components. Some precautions have been taken to avoid this, but the simplest and most effective method have been to redo the feature altogether.

5.1.3 Connecting hardware components to subnetworks

The framework limits the amount of ECUs connected to a CAN bus to five, in order to lower the busloads in these subnetworks. This promotes the generation of automotive architectures that contain feasible allocations. The distribution of ECUs over the set of subnetworks can therefore only be made uniformly if the number of ECUs is less than or equal to fives times the number of subnetworks. If this is not the case, each CAN bus is connected to five randomly chosen ECUs, and the remaining set of ECUs is distributed over the set of FlexRay subnetworks.

The connections between LIN buses and ECUs are randomly chosen, with every ECU having the same probability of being connected to. This means one ECU may be connected to several LIN buses. Each LIN is in turn connected to two sensors or two actuators from the same domain, unless the chosen domain only has one unconnected sensor or actuator left. Depending on required delivery times R_s of the signals transmitted by sensors or received by the actuators, the relatively low bit rate of LIN might not suffice. Since these signals must be transmitted over the bus, no matter how the software is allocated, the framework switches the LIN to a CAN if the bandwidth utilization of the signals are too high for the LIN. This check is done according to

Table 5.2: ***The setup for the two evaluated scenarios.*** *The parameters for optimization are the same for both scenarios, with the exception that the chromosome length m depends on the number of SWCs.*

Optimization parameters	
Archive size $\|P\|$	100
Number of generations G	1000
Tournament size n_{tour}	2
Tournament selection parameter p_{tour}	0.8
Swap mutation rate p_{swap}	$\frac{1}{m}$
Point mutation rate p_{point}	$\frac{1}{m}$

Generation parameters		
	Scenario 1	Scenario 2
Number of SWCs $\|\boldsymbol{C}\|$	17	250
Number of domains $\|\boldsymbol{D}\|$	2	6
Number of ECUs $\|\boldsymbol{E}\|$	4	25
Number of CAN $\|\boldsymbol{B}_{\text{CAN}}\|$	1	3
Number of FlexRay $\|\boldsymbol{B}_{\text{Flex}}\|$	1	2

Equation (3.2), where the period times T_s^b on LIN depend on the delivery time requirements, see Section 3.1. However, since the assignment of period times depend on the software allocation, the required values for T_s^b are not known during the generation procedure. For each signal transmitted over the LIN, the period time is estimated with $T_s^b = 20\,\text{ms} - \tau_s^b$, where τ_s^b is the transmission time over the LIN bus for signal s. If the resulting busload l_b in Equation (3.2) exceeds 100% of the capacity of LIN, a CAN replaces the LIN.

The remaining sensors and actuators are connected directly to the subnetworks. These connections are made probabilistically, with the probability of connecting to a certain subnetwork being proportional to the number of ECUs already connected to that subnetwork.

5.2 Evaluated scenarios

Given the framework presented in Section 5.1, the optimization was evaluated using two different scenarios, presented in Table 5.2 together with the parameters used for the optimizations. Ten automotive architectures were generated using each scenario, and every architecture was optimized ten times, in order to perform a more thorough performance assessment of the optimization method.

The first scenario describes small-scale automotive architectures that have approximately $6.7 \cdot 10^7$ different possible allocations. Despite their rather extensive search spaces, it was possible to perform exhaustive searches in reasonable amount of time, and thus find the true Pareto-front for each architecture. This meant that the optimization method could be evaluated with regard to the ground truth. The hypervolume ratio presented in Subsection 4.1.2 was used for evaluating all optimizations and if the true Pareto-front was found during the optimization, the performance evaluation was also performed in regard to the number of generations until it was found. The exhaustive search is also able to find the total number of feasible solutions as well as the worst feasible objective values. Respectively, these measurements give an indication of how constrained the optimization problem is and a suitable reference point for hypervolume computations.

Architectures described by the second scenario have enormous search spaces, which means that exhaustive searches and ground truth comparisons were not an option. Instead, the consistency of the approximated Pareto-front, with regard to multiple optimizations of the same architecture, was measured and used to evaluate the optimization method. This was done using hypervolume computations, with reference point as the worst feasible objective values found during the optimizations of that particular architecture. In order to yield a value for the hypervolume ratio without knowledge of the true Pareto-front, the set approximated Pareto-fronts from all optimizations were combined and the set of Pareto-dominant solutions in that set were selected. Since the total number of feasible solutions could not be computed, the ratio of feasible solutions was estimated by evaluating the feasibility of 10^5 randomly selected allocations.

The results from the described evaluations are presented in the following chapter. This chapter also features an investigation of a single optimization of a small-scale architecture, as well as brief presentations of two different solutions within the Pareto-front.

6
Results

This chapter presents the results from using the multi-objective genetic algorithm to optimize the allocation of software in automotive architectures. In order to acquaint the reader with the produced results, the first section provides a description of a small-scale automotive architecture, generated using the developed framework, as well as two of its Pareto-optimal software allocations. The second section compares the results from optimizations and exhaustive searches performed on ten small-scale automotive architectures. Lastly, the consistency of the optimization method is assessed through the results from optimizations of ten large-scale automotive architectures.

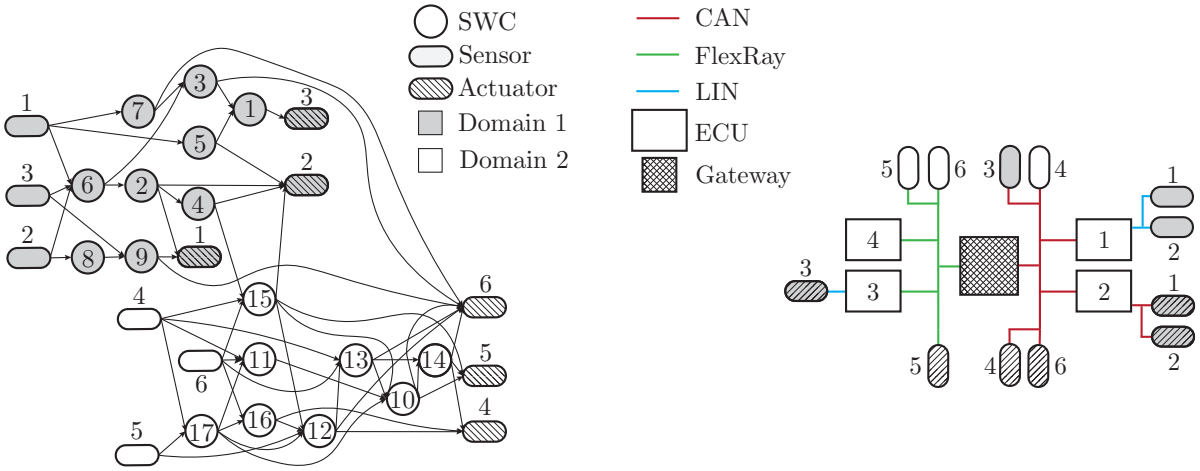
6.1 Investigation of an optimized automotive architecture

This section provides an example of the optimization of a small-scale architecture, generated according to scenario 1. How the signals are routed between sensors, SWCs, and actuators within the software architecture is visualized in Figure 6.1a. This architecture successfully mimics the desired transmission patterns, described in Subsection 5.1.1. A few inter-domain communications were generated, but most signals are transmitted within their respective domains. From this figure, it is hard to distinguish the separate features, but indications of signal chains can still be seen. Figure 6.1b illustrates the hardware architecture, which includes the sensors and actuators used within the software architecture, and adequately resembles the example shown in Figure 2.2.

An optimization of the SWC allocations was performed using the parameters presented in Table 5.2. Additionally, an exhaustive search of the solution space was carried out, and revealed that the ratio of feasible solutions for this architecture is 1.4% of all possible allocations. This means that with a initial population of 100 randomly generated individuals, one individual is likely to encode a feasible allocation. In Figure 6.2, the approximated Pareto-front after 1000 generations of the optimization is shown. This figure also presents the true Pareto-front and the reference point, given by the worst feasible value for each objective, both of which were acquired through the exhaustive search. Not all Pareto-dominant solutions were found during the optimization. Most of the solutions promoting busload were found, while the Pareto-optimal solutions which favored memory utilization seemed more difficult to obtain.

Table 6.1 presents the allocations corresponding to the lowest feasible value for each objective, together with the respective busloads and memory utilizations for each bus and ECU in the hardware architecture. To obtain the lowest f_{busload} for this architecture, the SWCs seem to be allocated according to domain. The largest ECU, e_2, has eight SWCs allocated to it, seven of which belong to domain 1. This enables a high ratio of the signals within domain 1 to avoid transmission over buses. On the other hand, with this allocation, only two of the four ECUs have memory utilizations within the optimal range of $e_{\text{util}} \in [0.6, 0.9]$. The allocation that yields the lowest memory utilization is presented in the right column of Table 6.1. This allocation leaves ECU e_1 empty, which results in high memory utilizations e_{util} for the remaining three ECU. The memory utilizations for e_3 and e_4 are still within the optimal range, but e_2 exceeds the optimal range by 6%.

Comparisons between the busloads of the two allocations, also presented in Table 6.1, reveal that while the allocation with the lowest f_{memory} has 22% and 31% more signals transmitted over the CAN and FlexRay buses respectively, the busloads l_b are increased by 59% and 45%. This implies that the allocation corresponding to the lowest f_{busload} favors intra-ECU transmission of the large-sized signals, rather than transmission over buses.



(a) ***Software architecture generated by framework.***

(b) ***Hardware architecture generated by the framework.***

Figure 6.1: ***Software and hardware architectures generated according to scenario 1.*** *The sensors and actuators shown in the two figures are the same.*

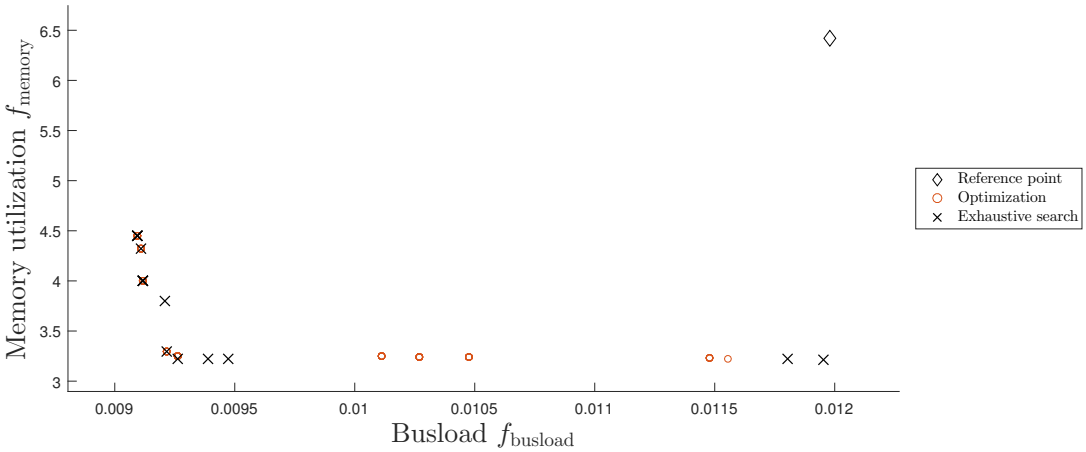


Figure 6.2: ***The Pareto-front obtained after single optimization of an architecture generated according to scenario 1.*** *The true Pareto-front, marked by crosses, was obtained through an exhaustive search of the solutions space. The Pareto-front approximated by the optimization, marked by circles, does not agree perfectly with the true Pareto-front. However, most of the Pareto-dominant solutions that favor busload were found during the optimizations. The reference point is set to the worst feasible value for each objective, which also was determined through the exhaustive search.*

Table 6.1: ***The allocation of SWCs for the solutions with the lowest value for the busload and memory utilization objectives respectively.*** $\boldsymbol{C}_e$ *denotes the set of SWCs alllocated to ECU* e, e_{util} *is computed according to Equation* (3.6), $|\boldsymbol{S}_b|$ *is the number of signals transmitted over bus* b, *and* l_b *is computed according to Equation* (3.2).

	Allocation with lowest f_{busload}		Allocation with lowest f_{memory}	
ECU	$\boldsymbol{C}_e$	e_{util}	$\boldsymbol{C}_e$	e_{util}
e_1 (322 kB)	c_7 (74 kB), c_{10} (99 kB)	0.54		0
e_2 (789 kB)	c_2 (70 kB), c_3 (114 kB), c_4 (88 kB), c_5 (90 kB), c_6 (75 kB), c_8 (105 kB), c_9 (110 kB), c_{15} (115 kB)	0.97	c_1 (64 kB), c_{10} (99 kB), c_2 (70 kB), c_7 (74 kB), c_8 (105 kB), c_{11} (75 kB), c_{12} (82 kB), c_{15} (115 kB), c_{17} (76 kB)	0.96
e_3 (407 kB)	c_1 (64 kB), c_{14} (93 kB), c_{16} (110 kB)	0.65	c_3 (114 kB), c_4 (88 kB), c_5 (90 kB), c_6 (75 kB)	0.90
e_4 (473 kB)	c_{11} (75 kB), c_{12} (82 kB), c_{13} (113 kB), c_{17} (76 kB)	0.73	c_9 (110 kB), c_{13} (113 kB), c_{14} (93 kB), c_{16} (110 kB)	0.90
Bus	$\lvert\boldsymbol{S}_b\rvert$	l_b	$\lvert\boldsymbol{S}_b\rvert$	l_b
CAN	18	0.0148	22	0.0235
FlexRay	16	0.00672	21	0.00976

6.2 Scenario 1

As described in Section 5.2, the performance of the optimization method was, in part, evaluated by performing ten independent optimizations, each running for 1000 generations, of the software allocation within a single architecture generated according to scenario 1. This procedure was thereafter repeated for, in total, ten such automotive architectures. The results from this evaluation are presented in Table 6.2, which also contains the true Pareto-front and ratio of feasible solutions, both obtained from exhaustive searches of each architecture's solutions space. The ratios of feasible solutions indicate that scenario 1 generates adequately constrained optimizations problems, with roughly $8.3{\cdot}10^5$ of the $6.4{\cdot}10^7$ possible solutions being feasible allocations.

Similarly to the results described in Section 6.1, several of the performed optimizations did not find the entire true Pareto-front. Averaged over all architectures, 48% of the optimizations were able to obtain every solution in the true Pareto-front within the 1000 generations. Only the optimizations of architectures 3 and 4 were able to consistently find every solution within the Pareto-front. This might be due to the fact that those Pareto-fronts contained the fewest number of solutions, namely 1 and 6 respectively. The Pareto-fronts corresponding to the remaining architectures contained between 7 and 22 solutions each, with overall mean being 9.8. As seen in Table 6.2, a higher number of solutions in the Pareto-front was generally related to a low ratio of optimizations that obtained the entire Pareto-front.

Out of the optimizations that were able to find the entire set of Pareto-dominant solutions, the average number of generations until this occurred was 360. The standard deviation was however significantly high, and for some individual optimizations, the entire Pareto-front was located after 900 generations had passed. For the optimizations that did not obtain the entire Pareto-front, 57% of the Pareto-dominant solutions were found on average. The standard deviation is however high for this measurement as well, with the ten optimizations of architecture 5 on average finding 22% of the Pareto-dominant solutions. These measurements indicate that the structure of the generated architecture effects the optimization method's ability to find the entire true Pareto-front.

Through the exhaustive searches, the true hypervolume $\mathcal{H}_{\text{true}}$ could be computed for each of the ten architectures, with the worst feasible value for each objective and architecture being used as reference point. This allowed the hypervolumes of the approximated Pareto-fronts to be evaluated through use of the hypervolume ratio $\mathcal{H}_{\text{ratio}}$. By computing the mean of $\mathcal{H}_{\text{ratio}}$ over all optimizations of the same architecture, and then averaging the resulting ratios over all architecture, the value of 98.9% was obtained. This means that even though less than half of the optimizations yielded the true Pareto-front, the quality of the approximated

Table 6.2: ***Results from ten optimizations and one exhaustive search of each of the ten architectures generated according to scenario 1.*** *Each optimization continued for 1000 generations.* * *denotes that the values were obtained through the exhaustive search,* ** *indicates that the average was taken over the optimizations that did obtain the true Pareto-front, and* *** *means that the average was taken over the optimizations that did not manage to find every Pareto-dominant solution.*

Architecture	1	2	3	4	5	6	7	8	9	10	$\mu \pm \sigma$
Ratio of feasible solutions*	0.056	0.013	0.002	0.003	0.033	0.002	0.021	0.014	0.019	0.014	0.013 ± 0.010
Number of Pareto-dominant solutions*	18	8	1	6	9	7	8	10	9	22	9.8 ± 6.0
Ratio of optimizations that obtained the true Pareto-front	0.2	0.6	1	1	0.5	0.7	0.8	0	0	0	0.48 ± 0.40
Mean number of generations until the true Pareto-front was obtained**	593	519	39	110	554	343	364	n/a	n/a	n/a	360 ± 217
Mean ratio of Pareto-dominant solutions found***	0.76	0.86	n/a	n/a	0.22	0.62	0.56	0.61	0.34	0.57	0.57 ± 0.21
True hypervolume $\mathcal{H}_{\text{true}}{}^{*}$	0.105	0.163	0.285	0.320	0.178	0.313	0.130	0.119	0.088	0.078	0.178 ± 0.094
Mean hypervolume ratio $\mathcal{H}_{\text{ratio}}$	0.989	0.998	1	1	0.993	0.999	0.999	0.998	0.933	0.980	0.989 ± 0.021

Pareto-fronts were overall high.

6.3 Scenario 2

The optimization procedure described in Section 6.2 was applied to ten architectures generated according to scenario 2, and the results from these optimizations are presented in Table 6.3. An important note is that the ten architectures included in this table are the ones for which the optimization procedure was able to find feasible solutions. In total, 15 architectures had to be generated in order to obtain ten architectures that the optimization successfully found feasible solutions for. Since there was no possibility of performing exhaustive searches for the five discarded architectures, it cannot be concluded whether there in fact exist feasible solutions or not. The estimation of the feasibility ratio, performed by evaluating 10^5 random allocations, yielded no feasible solutions for any of the 15 architectures. This result indicates that architectures from scenario 2 have tighter constraints compared to architectures from scenario 1.

For every architecture included in Table 6.3, feasible solutions were found during each of the ten optimizations. The mean number of generations until feasible solutions were found, averaged over all ten architectures, was 45. This means that, provided that feasible solutions were obtained during an optimization, they were found during the initial phase of the optimization. The approximated Pareto-fronts of the first architecture is illustrated in Figure 6.3. The lack of agreement between these fronts seems to indicates that the optimization method is less consistent for large-scale architectures than for small-scale architectures. This is emphasized by the mean hypervolume ratio $\mathcal{H}_{\text{ratio}} = 0.825$, averaged over all architectures and optimizations, which can be compared to $\mathcal{H}_{\text{ratio}} = 0.989$ for the small-scale architectures. The reason for less consistency could be that the number of generations was too low, since some optimizations were still improving their approximated Pareto-fronts. As described in Section 6.2, there was a high deviation in the number of generations passed before the true Pareto-front was found for the small-scale architectures. However, to provide a fair comparison between the optimizations, the same number of generations had to be used. In order to experience reasonable evaluation times, the number of generations was limited to 1000.

Table 6.3: ***Results from ten optimizations of each of the ten architectures generated according to scenario 2.*** *Each optimization continued for 1000 generations. The Pareto-front used to compute the true hypervolume $\mathcal{H}_{\text{true}}$ was determined by choosing the set of Pareto-dominant solutions from the combined set of approximated Pareto-fronts, see Figure 6.3.*

Architecture	1	2	3	4	5	6	7	8	9	10	$\mu \pm \sigma$
Mean number of generations until a feasible solution was found	38	55	40	39	54	66	36	32	54	35	45 ± 11
Hypervolume of combined Pareto-front $\mathcal{H}_{\text{true}}$	0.145	0.176	0.021	0.149	0.188	0.097	0.113	0.170	0.191	0.187	0.144 ± 0.054
Mean hypervolume ratio $\mathcal{H}_{\text{ratio}}$	0.760	0.900	0.836	0.761	0.821	0.854	0.847	0.815	0.822	0.830	0.825 ± 0.042

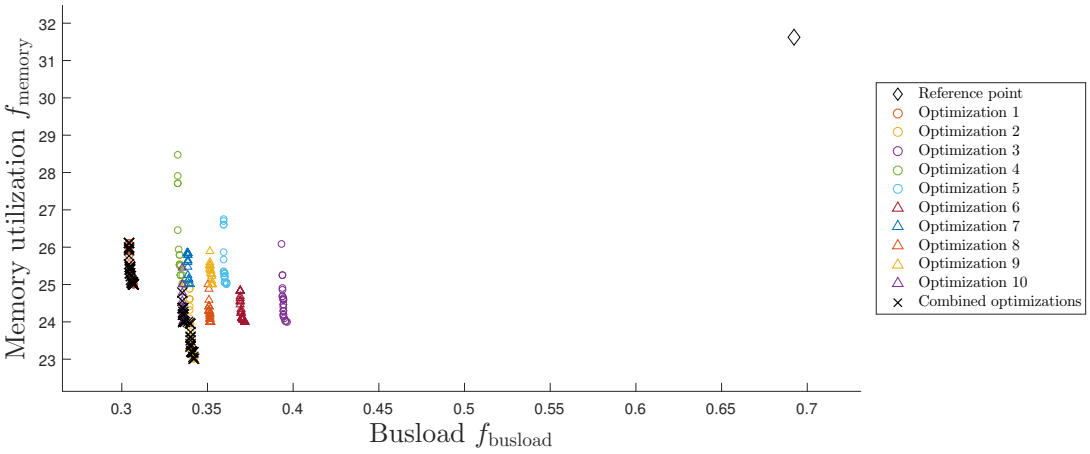


Figure 6.3: ***The Pareto-fronts approximated by ten different optimizations of an architecture generated according to scenario 2.*** *Since the large size of the architecture does not allow for an exhaustive search of the solution space, the true Pareto-front of the problem is not known. Each individual optimization, marked by either circles or triangles, could though be evaluated with regard to the performance of the other optimizations. These evaluations were performed with respect to the crosses in the figure, which illustrate the set of Pareto-dominant solutions in the combined set of approximated Pareto-fronts. The reference point was set to worst feasible value, found during the ten optimizations, for each objective.*

7
Discussion

This chapter summarizes the most important results and discusses these with respect to the developed and used methods. The main aim of this work was to develop and implement an MOGA in order to optimize automotive architectures. Whether this was successfully done or not is discussed in the first section, together with some of the limitations of the optimization. The following sections discusses the simulations environment and the generation framework, respectively. Lastly, some future improvements that can be made to extend this work are presented.

7.1 Using an MOGA to optimize software allocation

Section 6.2 shows that the developed optimization method is very successful in obtaining approximated Pareto-fronts of high quality for small-scale automotive architectures. Using hypervolume as measurement, the optimizations achieve a hypervolume ratio of $\mathcal{H}_{\text{ratio}} = 0.989$ on average. The reason for not achieving $\mathcal{H}_{\text{ratio}} = 1$, i.e. finding the true Pareto-front during every optimization, may be due to premature convergence. More or less every performed optimization had converged before 1000 generations had passed, but the true Pareto-front was only found in 48% of the cases. One possible way of reducing the likeliness of premature convergence can be to investigate different sets of optimization parameters. As indicated by the results obtained for large-scale architectures, the set of parameters used in this work seems to perform well during the initial phase of the optimization, during which the algorithm searches for feasible solutions. However, the subsequent optimization of the Pareto-front could benefit from altered parameters, in order to maintain a wider range of genetic material within the population. One possible alteration is to increase the probability of swap mutation once feasible solutions have been found. Since swap mutations do not change the number of SWCs allocated to each ECU, thus keeping the memory utilization of ECUs more or less equal, this alteration might result in a more efficient investigation of the feasible solution space.

The purpose of performing optimizations of large-scale architectures, for which the results are presented in Section 6.3, was to assess the consistency of the optimization method. In contrast to the convergence that occurred when optimizing small-scale architectures, the optimizations of large-scale architectures were still adjusting their Pareto-fronts when 1000 generations had passed. But even considering this observation, the fact that the approximated Pareto-fronts differed, as illustrated in Figure 6.3 and by the average hypervolume ratio of $\mathcal{H}_{\text{ratio}} = 0.825$, implies that the optimization method is inconsistent for large architectures. Just as for the small-scale architectures, this may be due to premature convergence. With the enormous search space of scenario 2, the allocations encoded in the first feasible solutions may heavily affect the continuous evolution of the population. Further clarification may come from investigating the genetic diversity in the population and which parts of the solution space are explored during different optimizations.

Unfortunately, it is difficult to determine whether the five generated large-scale architectures, for which the optimization method did not manage to obtain any feasible solutions, do in fact possess any feasible allocations. As explained Section 6.3, the average number of generations until a feasible solution was found was 45. While this result indicates that the optimization method is very capable of finding feasible solutions, and thus implies that the five mentioned architectures do not possess any feasible solutions, these architectures pose questions regarding the validity of the optimization method. One way of decreasing the amount of uncertainty is to use real automotive architectures when evaluating the optimization method. By using the already implemented software allocation as a comparison for the optimized counterparts, there is a guarantee that feasible solutions exist. Additionally, it becomes far easier to evaluate the optimized allocations, and thus concluding whether the optimization method has merit within the automotive industry.

7.1.1 Application to real-world architectures and the curse of dimensionality

Even though the optimization seems to perform well for small-scale architectures, one must keep in mind that the true Pareto-front only constitute the best allocations according to the objective measures defined in the simulation environment, see Chapter 3. The Pareto-dominant solutions would thus not necessarily perform the best if they were applied to a real-world vehicle, which contains several aspects that have been neglected in this work. The flaws and possible improvements of the simulation environment are discussed in Section 7.2, but how the complicated nature of automotive architecture affects the validity of using MOGAs when optimizing software allocations is also worth commenting. Is it feasible to compress, for example, the busload of every bus in the hardware architecture into a single scalar value f_{busload}, or will such a compression inevitably fail to capture the interconnectivity and dependencies of the hardware network?

One approach to deal with complicated relations within and between different objectives is to introduce additional objective functions, which allows for less compression of information. In doing this, one must though consider *the curse of dimensionality* [45], which states that as the number of objectives increases, Pareto-dominance becomes rarer and the number of Pareto-optimal solutions increases. This is one of the biggest problems within MOGAs, as it limits their application to problems with several objectives [45]. Another problem related to an increased number of Pareto-optimal solutions is that the vehicle manufacturer eventually must choose one of these for implementation, and the process of going through and understanding the implications of each solution is a difficult and time-consuming task.

7.2 The design and usage of the simulation environment

As mentioned in Subsection 7.1.1, how well the simulation environment is modeled constitutes a contributing factor to the validity of using MOGAs to optimize automotive architectures. Regarding the objective functions used in this work, a few interesting remarks can be made based on the allocations presented in Table 6.1. The allocation that yields the best memory utilization leaves ECU e_1 empty and thereby exceeds the optimal memory utilization range for e_2. At first glance, this may not seem to be a particularly good utilization of the available ECU memory, due to the high memory utilization of e_2. However, since the weight function for memory utilization takes the possible removal of ECUs into account, the benefit from removing e_1 outweights the drawback from exceeding 90% memory utilization for e_2. This would although not have been the case if the weight function, see Figure 3.2b, would have had a more steep slope outside of the optimal range. The angle of the slope thus determines the relative importance of removing an ECU. Since a benefit of multi-objective optimization is the possibility of avoiding this kind of weighting parameters, the objective function for memory utilization could have benefited from being divided into a cost objective, which models the possible removal of hardware components, and a pure memory utilization objective. These results illustrates the importance of carefully formulating the models within the simulation environment and how changes to a model may greatly affect the outcome of an optimization.

While the concept of a simulation environment, which can assess the performance of individual components as well as entire automotive architectures, seems to be a practical tool for evaluation, constructing one is easier said than done. Large teams of system architects and engineers are needed to ensure that performance demands, such as signals being delivered on time, are fulfilled. The work is often done iteratively between development and testing, and there are also requirements for flexibility since new functionalities may be added during the development of the vehicle. It may be feasible to maintain models for each type of entity within the architecture, but replacing the current methods for evaluating entire automotive architectures with a simulation environment would require far more work than have been performed in this thesis.

7.3 The realism and feasibility of the generated architectures

Due to the limited amount of available data regarding real automotive architectures, it is hard to conclude whether the hardware and software architectures generated by the developed framework, described in Chapter 5, are in fact realistic. With more available data, it would be valuable to develop an evaluation method which compares the generated architectures with real data and gives a measurement of their similarity. This was however outside the scope of this work.

Regardless of the uncertainties whether the produced data is realistic or not, the framework appears to generate architectures that pose complicated optimization problems. The small-scale architecture visualized in Figure 6.1 provides an illustrative example of this. The software allocation that yields the lowest busload, presented in the left column of Table 6.1, has all but two of the SWCs in domain 1 allocated to ECU e_2, which has the highest memory capacity of all ECUs. As described in Section 6.1, this successfully lowers the busloads in the hardware network, since most of the signal exchanges in domain 1 occurs inside e_2. However, by inspection of Figure 6.1a, it would seem that allocating the SWCs of domain 2 to e_2 would lower the busload even more, since more intra-domain signals are transmitted within this domain. On the other hand, the sensors and actuators within domain 1 are more closely connected to e_2 than the sensors and actuators within domain 2 are, which makes the first mentioned allocation preferable. This demonstrates that finding the best allocations of even a small-scale architecture is not an intuitive problem.

The current framework is rather limited by the static parameters presented in Table 5.1 and the input parameters used for scenarios 1 and 2. These parameters were chosen in order to generate as many architectures with feasible allocations as possible, but even these scenarios cannot guarantee feasibility. Using different parameters may either increase the risk of generating architectures where no feasible allocations exists, or result in less interesting optimization problems. One future improvement of the generation framework can be to ensure that architectures without feasible allocations cannot be generated. Some precautions, such as switching highly loaded LIN buses to CAN buses, were taken in this work in order to remove apparent sources of error, but these were not enough. In order to make this improvement, more thorough investigations of the causes on infeasibility first need to be made and then solved accordingly. An alternative approach can be to lower the amount of stochastic elements in the generation, and in that way have more control over what is generated. However, such restrictions on the generation may lower the diversity between different architectures, and alterations within the framework should be made with caution.

From an academic point of view, a framework for generating diverse architectures is valuable in order to investigate and evaluate different optimization methods. The development of a universal framework, used for both benchmarking and standardization, could greatly benefit the research field of optimization of software allocation in vehicles.

7.4 Future work

A recurring topic in the discussions above has been the need to test and compare the implemented methods with data from vehicle manufacturers. The inability to apply the developed optimization method to real-world architectures makes it hard to evaluate the merit of this kind of study for industrial usage. Therefore, future work could be to investigate possible methods for extracting data from vehicle manufactures in formats suitable for optimization.

For the simulation environment, future work can be to develop more models of the dynamics within separate components, similar to [14] for CAN buses and [27] for FlexRay buses. Such models can be used as assistance when evaluating whether a software allocation is feasible, rather than serve as basis for optimization. If the simulation environment is to be used for optimization, as a tool that evaluated entire automotive architectures, the objective formulations have to be revised in order to assess the viability of compressing the multitude of measurements into scalar values.

This work has mainly focused on the approximated Pareto-front obtained at the end of an optimization, and only provided brief descriptions of the implications that these allocations have on the automotive architecture. More extensive studies of the similarities and differences between these allocations, for example analyzing whether some SWCs generally are allocated to certain ECUs, will alleviate the process of improving the optimization method. To investigate the possible causes of the premature convergence, the allocations can be studied during the course of an optimization, to understand what leads to local optima and which countermeasures can be taken to avoid it. One solution can be to introduce a dynamic mutation rate, which depends on the genetic diversity of the population.

8

Conclusion

Pareto-fronts for vehicle software allocation have been approximated within both small-scale (17 SWCs and 4 ECUs) and large-scale (250 SWCs and 25 ECUs) artificial automotive architectures. By comparing the approximated Pareto-fronts from 100 optimizations of 10 different small-scale architectures to the corresponding true Pareto-fronts, an average hypervolume ratio of $\mathcal{H}_{\text{ratio}} = 0.989$ was obtained. It is shown that the problem-adapted version of NSGA-II is able to find the true Pareto-front during 48% of these optimizations.

The 100 optimizations performed on 10 different large-scale architectures produce approximated Pareto-fronts of varying quality, which indicate that the optimization method behaves inconsistently for larger architectures. Furthermore, the optimization method was unable to obtain feasible allocations for five artificial architectures. Comparisons with other optimization results indicate that these architectures do not possess any feasible solutions, but it may instead be due to limitations of the optimization method.

Artificial automotive architectures have been generated through a developed framework, due to limited availability of real data for automotive architectures. The difficulty in applying the optimization method to real automotive architectures poses a problem when evaluating its industrial merit. The potential of facilitating the software allocation at Volvo Cars, through the use of this method, thus currently appears to be limited.

Bibliography

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 33–42.

[2] P. Pelliccione, E. Knauss, R. Heldal, M. Ågren, P. Mallozzi, A. Alminger, and D. Borgentun, "A Proposal for an Automotive Architecture Framework for Volvo Cars," in *Automotive Systems/Software Architectures (WASA), 2016 Workshop on*. IEEE, 2016, pp. 18–21.

[3] I. Moser and S. Mostaghim, "The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE, 2010, pp. 1–8.

[4] B. Hardung, "Optimisation of the allocation of functions in vehicle networks," Ph.D. dissertation, Friedrich-Alexander University Erlangen-Nürnber, Erlangen, Germany, November 2006.

[5] AUTOSAR Consortium, "Automotive open sytem architecture (autosar)," http://www.autosar.org.

[6] V. S. Honnavara, "Cost optimization by method of allocating software component units to electronic control units for model-driven designs," Master's thesis, North Carolina State University, Raleigh, NC, United States, 2008.

[7] R. E. Korf, "A new algorithm for optimal bin packing," in *AAAI/IAAI*, 2002, pp. 731–736.

[8] M. Dohr and B. Eichberger, "Guided mutation strategies for multiobjective automotive network architecture," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*. IEEE, 2013, pp. 2473–2479.

[9] M. Zeller and C. Prehofer, "Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems," *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1067–1082, 2013.

[10] D. Thiruvady, I. Moser, A. Aleti, and A. Nazari, "Constraint programming and ant colony system for the component deployment problem," *Procedia Computer Science*, vol. 29, pp. 1937–1947, 2014.

[11] Real-Time Systems Laboratory, "An introduction to AUTOSAR," https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf, accessed: 2017-04-24.

[12] K. Tindell, A. Burns, and A. J. Wellings, "Calculating controller area network (CAN) message response times," *Control Engineering Practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[13] C. Smith, *The Car Hacking Handbook*. San Fransisco, CA, United States: No Starch Press, 2016.

[14] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[15] J. Happian-Smith, *An introduction to modern vehicle design*. Elsevier, 2001.

[16] LIN Consortium, "LIN specification package, revision 2.0," *Munich, Germany*, 2003.

[17] CAN Specification, "Version 2.0," *Robert Bosch GmbH*, 1991.

[18] FlexRay Consortium, "FlexRay communication systems protocol specification, version 3.0. 1 [OL]," 2010.

[19] S. C. Talbot and S. Ren, "Comparision of fieldbus systems can, ttcan, flexray and lin in passenger vehicles," in *Distributed Computing Systems Workshops, 2009. ICDCS Workshops' 09. 29th IEEE International Conference on*. IEEE, 2009, pp. 26–31.

[20] "FlexRay Automotive Communication Bus Overview," National Instruments, Tech. Rep., 08 2016. [Online]. Available: http://www.ni.com/white-paper/3352/en/

[21] "Controller Area Network (CAN) Overview," National Instruments, Tech. Rep., 08 2016. [Online]. Available: http://www.ni.com/white-paper/2732/en/

[22] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[23] J. Cook and J. Freudenberg, "Controller Area Network (CAN)," 2007.

[24] R. DeMeis, "Cars sag under weighty wiring," *Electronic Times*, vol. 10, p. 24, 2005.

[25] "Endres". ("2014") "CAN-Frame mit Pegeln mit Stuffbits.svg". [Online]. Available: "https://commons.wikimedia.org/wiki/File:CAN-Frame_mit_Pegeln_mit_Stuffbits.svg"
[26] "Introduction to the Local Interconnect Network (LIN) Bus," National Instruments, Tech. Rep., 08 2016. [Online]. Available: http://www.ni.com/white-paper/9733/en/
[27] M. Grenier, L. Havet, and N. Navet, "Configuring the communication on FlexRay-the case of the static segment," in *4th European Congress on Embedded Real Time Software (ERTS 2008)*, 2008.
[28] M. Dohr and B. Eichberger, "Evolutionary Routing Strategies for Automotive Networks," in *Proceedings of the International Conference on Genetic and Evolutionary Methods (GEM)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 1.
[29] E. F. Moore, "The shortest path through a maze," in *Proc. International Symposium on the Theory of Switching*. Harvard University Press, 1959, pp. 285–292.
[30] C. Y. Lee, "An algorithm for path connections and its applications," *IRE transactions on electronic computers*, no. 3, pp. 346–365, 1961.
[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.
[32] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
[33] R. Olaechea, D. Rayside, J. Guo, and K. Czarnecki, "Comparison of exact and approximate multi-objective optimization for software product lines," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM, 2014, pp. 92–101.
[34] D. A. Van Veldhuizen, "Multiobjective evolutionary algorithms: classifications, analyses, and new innovations," DTIC Document, Tech. Rep., 1999.
[35] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.
[36] J. H. Holland, "Adaptation in natural and artificial systems. An introductory analysis with application to biology, control, and artificial intelligence," *Ann Arbor, MI: University of Michigan Press*, 1975.
[37] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, MA, United States: MIT press, 1998.
[38] M. Wahde, *Biologically inspired optimization methods : an introduction*. Southampton, United Kingdom: WIT Press, 2008.
[39] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety*, vol. 91, no. 9, pp. 992–1007, 2006.
[40] N. Srinivas and K. Deb, "Muiltiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary computation*, vol. 2, no. 3, pp. 221–248, 1994.
[41] E. Zitzler, M. Laumanns, L. Thiele *et al.*, "SPEA2: Improving the strength Pareto evolutionary algorithm," in *Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, 2001, pp. 95–100.
[42] R. Datta and K. Deb, *Evolutionary constrained optimization*. Springer, 2015.
[43] K. Deb, "Multi-objective optimization using evolutionary algorithms, 2001," *Chicheter, John-Wiley.*, 2001.
[44] P. Emberson, "Searching for flexible solutions to task allocation problems," Ph.D. dissertation, University of York, 2009.
[45] S. Kukkonen and J. Lampinen, "Ranking-dominance and many-objective optimization," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. IEEE, 2007, pp. 3983–3990.