Functionality assignment to partitioned multi-core architectures

Florin Maticu
s131084

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

An embedded system is a microprocessor based system that usually performs a predefined set of tasks related to monitor or control of an electrical/mechanical equipment. Given the increased complexity of the embedded systems both in software and hardware, functionality assignment becomes difficult to deal with when safety, schedulability, efficient use of hardware resources and communication networks are required.

The thesis focuses on software functionalities as defined in AUTOSAR[1] which is an automotive standard for Electronic Control Unit (ECU) software development. The goal of AUTOSAR (AUTomotive Open System ARchitecture) is to establish a standardized model of development that makes it possible for software developers to create reusable, safety and hardware independent software components.

The objective of this thesis is to propose an algorithm based on simulated annealing heuristic for the mapping of functions with different safety integrity levels onto integrated architectures composed of multi-core systems. We will consider an architecture composed of a multi-core ECUs running AUTOSAR OS with fixed-priority preemptive scheduling.

A mapping tool has been developed that takes as input an application model of the AUTOSAR software components and an architecture model of multi-core ECUs and determines:

- A mapping of *software components* to ECUs.
- A mapping of *functionalities* or AUTOSAR *runnables* to ECU cores.
- A mapping of *functionalties/runnables* to AUTOSAR *Os-Tasks*.
- A mapping of *Os-Tasks* to AUTOSAR *Os-Applications*.

[1] `http://www.autosar.org/`

Such that:

- It minimizes the overall communication bandwidths.
- It minimizes the variance of the core utilizations on the system.
- Functions with different safety integrity levels are spatial and temporal isolated.
- All the constraints regarding schedulability or provided by the software/system developer are met.

The proposed approach is evaluated on an automotive case study from Volvo Advanced Technology & Research in Götheborg, Sweden.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with mapping of mixed safety integrity functionalities on multi-core architectures running AUTOSAR OS.

The work has been supervised by Associate Professor Paul Pop.

Lyngby, 26-June-2015

Florin Maticu
s131084

Acknowledgment

I would like to thank my supervisor, Associated Professor Paul Pop, for his support, availability and advice throughout the thesis. Many thanks to Axbrink Christian from Volvo for helping me with the automotive use case.

Last but not least, I would like to thank my family and friends for their moral support and trust in me.

Contents

Abbreviations

Abbreviation	Meaning
AUTOSAR	Automotive Open System Architecture
ASIL	Automotive Safety Integrity Level
IOC	Inter OS-Application communicator
COM	Comunication Stack
ECU	Electronic Control Unit
RTE	Runtime Environment
OS	Operating System
VFB	Virtual Function Bus
WCET	Worst Case Execution Time

Chapter 1

Introduction

An embedded system is a microprocessor based system that usually performs a predefined set of tasks related to monitor or control of an electrical or mechanical equipment.

Often, an embedded system has to operate in an environment where it has to meet real-time constraints. We refer to such systems as real-time systems. [But04] defines a real-time system as "computing systems that must react within precise time constraints to events in the environment". Having real-time constrains means the "correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced[Kop97]". Depending on the consequences of missing a deadline, real-time systems can be classified in soft and hard real-time. Soft real-time systems can miss the deadlines once in a while, as the system will still function, but with degraded service, whereas in hard real-time systems, missing a deadline will lead to the failure of the system. A special class of real-time systems are safety critical where a failure (hardware or software) may result in loss of human life or environment hazards.

Most embedded systems are controlled by an Real-time Operating System (RTOS). The unit of execution in any OS is called process or task. A set of instructions that are executed sequentially on a processor are encapsulated inside a task or a process. In this thesis we will refer to the periodic tasks that are cyclically executed at specific moment in times. The OS allocates each process/task to the processor based on a scheduling algorithm. [But04] describes a number of scheduling algorithms for periodic tasks such as Rate Monotonic (RM), Earliest Deadline First (EDF) and Deadline Monotonic (DM). Given a set of tasks and a scheduling algorithm, the schedulability analysis checks if the tasks are schedulable on a processor such that all are meeting their deadlines.

The schedulability analysis has been presented by [LL73] and is based on the concept of processor utilization factor. The fraction of time spend by the processor in executing a task defines its utilization factor. The eq. (1.1) describes the processor utilization for "n" tasks where C_i represents execution time without interruption on a processor and T_i is defined as period of the task.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1.1}$$

The authors have also proved that there is a maximum value for the processor utilization bellow which a set of tasks are schedulable and a maximum value above which the tasks are not schedulable.
Regarding multi-core and schedulability, the authors of [PPEP08] have also shown how complex hierarchical scheduling policies, similar to AUTOSAR's, can be analyzed.

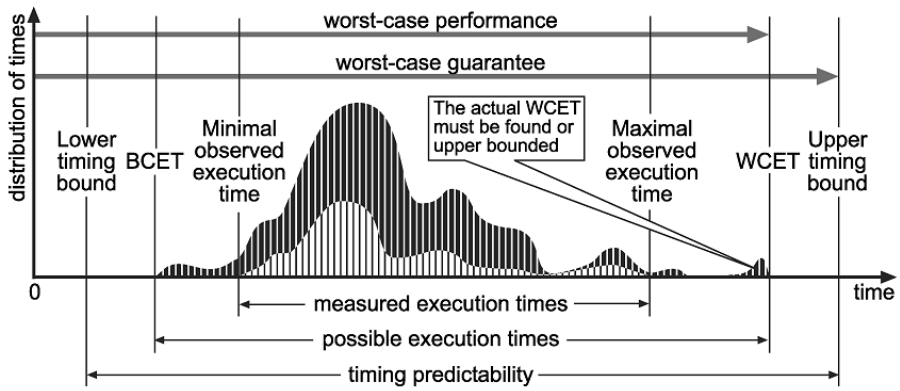


Figure 1.1: Execution times of a task [WEE+08]

Typically, a timing analysis for a task tries to determine the following values:

- *worst-case execution time* (WCET) which is the longest execution time when the task will run on a target processor.
- *best-case execution time* (BCET) which is the shortest execution time when the task will run on a target processor.
- *average-case execution time* (ACET) which is computed based on the execution time distribution of the task and typically the value is between BCET and WCET.

The schedulability analysis needs reliable task execution time in order to verify if a system works even in the worst cases possible. The execution time of a task varies with some probability over a range of time values and can not be determined accurately. Variations in execution times are influenced by the input data, compiler or the processor architecture. The fig. 1.1 presents different measured execution times (lower gray curve) along with probability distribution (dark curve) relatively to BCET and WCET. Due to an increase in size and complexity of the software for embedded systems, an exhaustive exploration of all possible executions to determine the worst and best execution times is hard or even impossible. Furthermore, new processor architectures with multiple cores, memory caches, pipeline stages, branch prediction, etc, are making the analysis of the timing even harder. According to [WEE+08] many measurements of execution times of a task overestimate the BCET and underestimate the WCET. For hard real-time systems we are concerned most with WCET analysis. It is required that the WCET estimate should be safe (the value must be close to the maximal observed execution time) and tight (little or no overestimation compared to the maximal observed execution time). To determine the execution

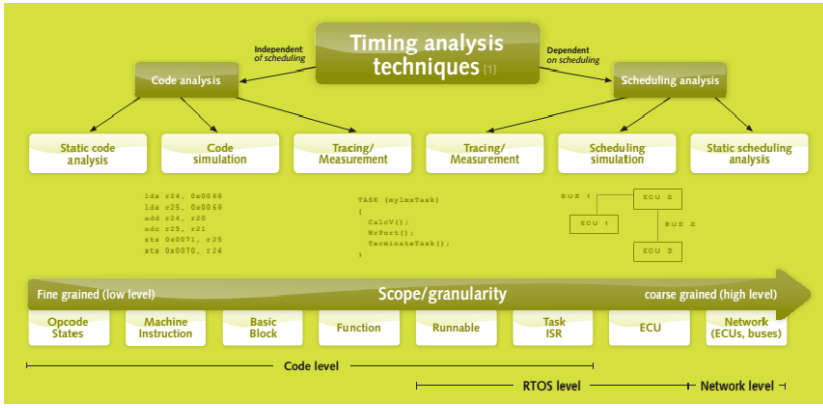


Figure 1.2: Timing analysis techniques, source:[Gli13]

times of a task, different timing analysis techniques can be applied and they are summarized in fig. 1.2:

- *Static code analysis* technique is based on reading and analyzing the source or binary code and provides a BCET and WCET value. This is a pure model approach and is not taking into the consideration the hardware architecture where the code runs.

- *Code simulation analysis* technique simulates the execution of a binary code for a given processor architecture and provides a WCET.

- *Static scheduling analysis* technique takes as input an application model and a scheduling algorithm and checks if any deadlines will be missed.

- *Scheduling simulation analysis* technique is similar to *static scheduling analysis* with the difference that it simulates the running of tasks and generates traces of their executions that can be analyzed further.

- *Measurement analysis* technique involves running the tasks on an embedded system within the OS and measuring their execution times using some hook functions provided by the OS.

- *Tracing analysis* technique involves capturing related events of the running system, adding a time stamp and save them into a buffer that can be analyzed off-line.

The increasing number of functionalities that are being implemented in a vehicle demands high-quality, reusable and safety critical hardware and software components. Nowadays, a modern vehicle has more than one hundred ECUs (Electronic Computational Unit) together with software that runs on it that provides different functionalities. An ECU (fig. 1.3) is an embedded system that has a microprocessor, memory and a number of I/O that controls one or more subsystems in a vehicle (engine, seat, door, etc). Inside the vehicle (fig. 1.4), the ECUs communicate through standardized network buses like CAN, FlexRay, etc.

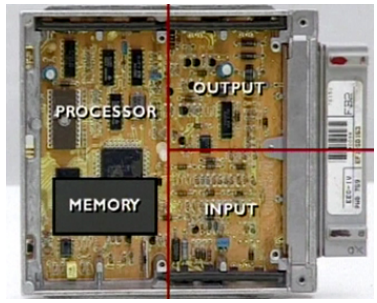


Figure 1.3: ECU[1]

The current trends for safety-critical applications are moving from federated architectures, where one function is implemented in one ECU, to integrated architectures, where several functions share resources on a single ECU.

[1] `http://www.cdxetextbook.com/images/efiSys_ECU.jpg`
[2] `http://www.embedded.com/print/4011425`

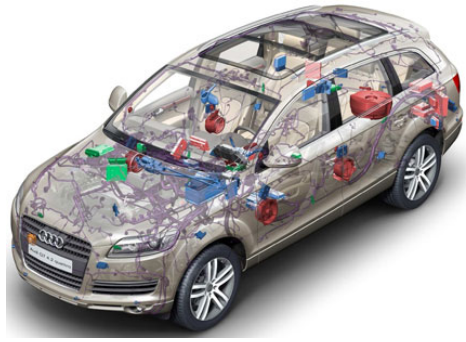


Figure 1.4: ECUs interconnected inside a vehicle[2]

In addition, multi-core ECUs are being adopted because of better cost, power consumption, size, fault-tolerance and performance. Increased complexity in functionality and hardware systems used in the automotive industry implies that more effort is added in the designing, developing and testing of the software functionalities. System developers have to map the functionalities to different ECUs such that safety according to ISO 26262[3] standard, schedulability of tasks running on different cores and bus bandwidth constraints are being met. The mapping task is not a trivial one and in many cases it can not be constructed "by hand". Therefore, tools are required to assist a system developer taking the decisions necessary for having a running system that meets all the constraints.

A method and a tool for assigning AUTOSAR functionalities called *runnables* to a distributed network of multi-core ECUs is proposed in the thesis. The goals are to minimize the overall bus bandwidth and the core utilization variance while all the constraints related to safety and schedulability are met.

Chapter 1 introduces the reader to the AUTOSAR framework and functional safety concepts related to automotive area. In Chapter 2 the application and the architecture model for the mapping problem is defined. Chapter 3 describes the mapping algorithm proposed and the use cases for testing. In chapter 4 the conclusions and possible future work are presented.

[3] `http://www.iso.org/iso/catalogue_detail?csnumber=43464`

1.1 Related Work

The authors of [SCCM15] propose an ILP (integer linear programming) approach for mapping AUTOSAR functions on a multi-core ECU to minimize the inter-core communication and balance the core load. They only consider one multi-core ECU and the applications have quite small number of functions which allows the ILP algorithm to find the optimal solution.

An approach for mapping AUTOSAR functionalities on multi-core ECU that takes into consideration timing and precedence constraints is proposed by the authors of [FLSN14]. Like in the previous article, they only consider one multi-core ECU. The authors are using an interesting heuristic method called systematic memory based simulated annealing (SMSA) and argue that it provides better results than the classic simulated annealing.

Another approach for mapping AUTOSAR functions into one multi-core ECU is presented in [NMBSL10]. Their goal was to balance the core utilization and the strategy used was to cluster functions that are exchanging data signals and iteratively assign them to the least loaded core. For the automotive applications (like the use case provided by Volvo) where most of the functions are exchanging signals, applying this strategy might result in most of them being clustered together and assigned to one core.

In [WMM+13], the authors proposed a mixed integer linear programming and a genetic algorithm for mapping AUTOSAR functionalities on a architecture composed of several single core ECUs. The goal is to optimize end-to-end timing response and the memory consumption. This paper is close to the mapping approach developed in this thesis with the difference that we are taking into consideration the safety level of AUTOSAR functions and our proposed architecture consists of multi-core ECUs. However, we ignore the end-to-end timing responses and the memory consumption.

1.2 Functional safety in Automotive

The growing complexity in the automotive industry makes the process of testing and validating safety related systems difficult. ISO 26262[4] has been adopted to provide an unifying standard regarding safety for all automotive electrical and electronic (E/E) systems.

[4] http://www.iso.org/iso/catalogue_detail?csnumber=43464

The standard provides a set of rules for safe product creation starting from the concept definition, development, production, operation and service. This paper focuses on functional safety and the risk classes (ASILs) associated with each software component. The Automotive Safety Integrity Level (ASIL) quantifies

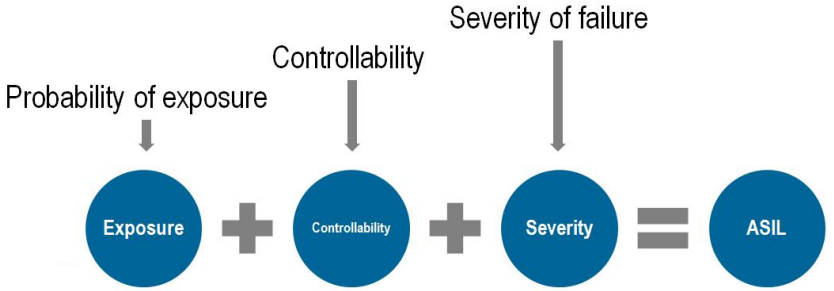


Figure 1.5: ASIL level estimation[5]

the risk of possible hazards when a failure happens at hardware and software level. In the case of road vehicles, the ASIL level for each software functionality is assigned based on the probability of exposure, how easily the failure can be handled by the driver and how severe the impact towards the road users will be if it happens (fig. 1.5). According to ISO 26262 there are five levels of ASIL:

- *QM (Quality Managed).* In the case of a failure, safety is not an issue but customer satisfaction is.
- *ASIL A.* In case of a failure, operational limitations are expected with no severe outcome.
- *ASIL B.* Ordinary driver can recover most of the time, usually no severe outcome.
- *ASIL C.* A good driver can recover (e.g. one brake works). In this case severe outcomes might be expected such as fatal crash.
- *ASIL D.* No driver is expected to recover (e.g. both brakes fail). Extremely severe outcomes expected such as multiple crash.

The ASIL level is determined for each functionality at the beginning of development process and depending on the assigned level, ISO 26262 provides different requirements for implementation and testing. In this thesis we do not address the issue of fault-tolerance. We assume that the required redundancy has been added to the system, for example, using a method proposed in [IPEP06].

[5] `http://www.ni.com/white-paper/13647/en/`

1.3 AUTOSAR

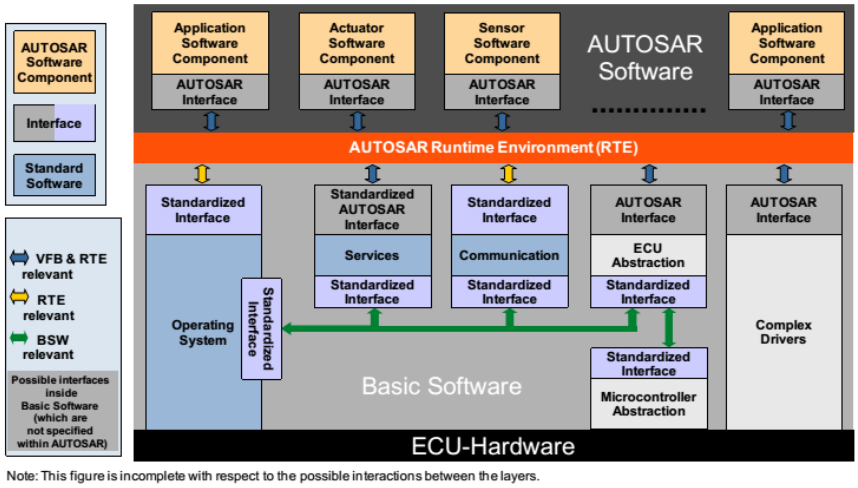


Figure 1.6: AUTOSAR layers, source:[AUT14a]

The AUTOSAR (AUTomotive Open System ARchitecture) is a partnership between vehicle manufactures, suppliers, hardware and software providers to develop a common standard that makes possible the development of reusable software functions for vehicles. AUTOSAR also focuses on performance, safety and provides a framework that can manage the growing complexity of electrical/electronic components used in a vehicle. The AUTOSAR framework has a standardized layered software architecture made of three parts fig. 1.6:

- An application software layer.
- A middle layer called Runtime Environment (RTE).
- The Basic Software layer (BSW).

The application layer is composed of the *software components* that provide the functionality required on the ECU.
The RTE layer defines a standardized application program interface (API) that allows an application to call a service from the Basic Software Layer. Furthermore, the communication between *software components* is also performed via the RTE layer.

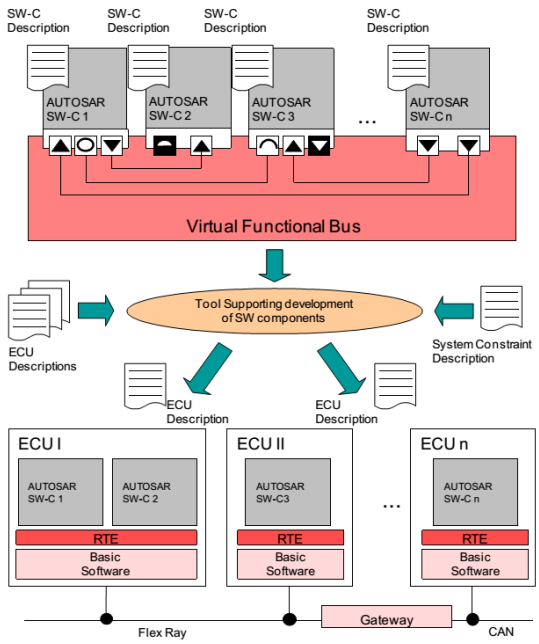


Figure 1.7: "Configure System" activity in AUTOSAR, source:[AUT14a]

The Basic Software Layer consists of the Operating System and modules that provide services like communication over a network, I/O, memory access. It provides an abstract layer to the *software components* that hides the ECU-hardware details.

The AUTOSAR methodology defines the development steps that allow the full configuration of an ECU starting from an application model and a system topology specification fig. 1.7. In AUTOSAR, an application model is composed of several *software components* that logically interact through a Virtual Function Bus (VFB). The Runtime Environment (RTE) can be seen as the implementation of the (VFB) providing the API necessary for the "logically" connected *software components* to exchange data signals and access OS services.

A system topology consists of a number of interconnected ECUs. In "Configure System" activity, first the *software components* are mapped to different ECUs. Once they are assigned to an ECU, the virtual connections between them are mapped to intra-ECU or inter-ECU network communications. Next, primitive data elements are mapped to signals. In the case of complex data types such as arrays and records, they are mapped into signal groups.

At the end, ECU configuration is performed by generating the Run-time Environment (RTE) and setting the Operating System and Basic Software modules used. One important Basic Software module is the Communication Stack (COM) that needs to be configured for inter-ECU communication. The fig. 1.7 presents an example of the outcome of the "Configure system" activity where a number of "n" software components logically connected through a VFB are mapped into "n" ECUs. After mapping and configuration, each ECU has it's own RTE layer and Basic Software generated.

1.3.1 Software components

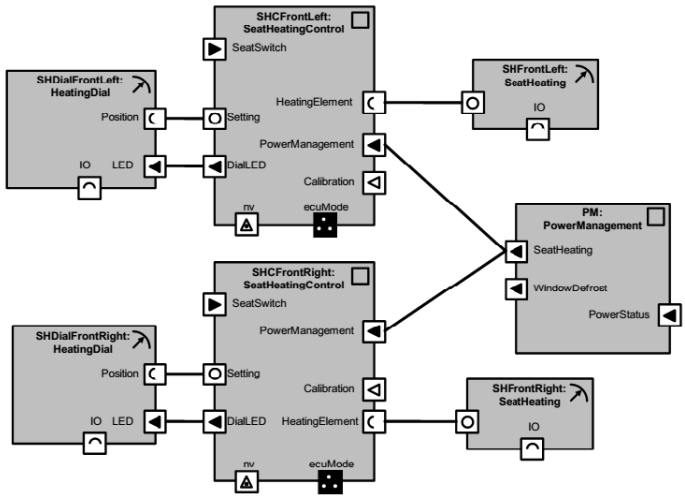


Figure 1.8: Seat heating application, source:[AUT14a]

An AUTOSAR application is composed of a several *software components* that interact with each other through well defined *ports*. The implementation of a *software component* does not depend on a particular hardware or type of communication needed for sending signals, therefore it can be relocated and run on different ECUs. An example of an application that controls the seat heating in the vehicle, composed of seven *software components* is presented in fig. 1.8.

Each *software component* requires the specification of the ports for communication with other components and the implementation of their internal behavior.

A *port* has a associated *port-interface* that is the *"contract"* between *one software component* that provides the interface (P-ports) and the one that requires an interface (R-Ports), fig. 1.9. There are different types of port interfaces

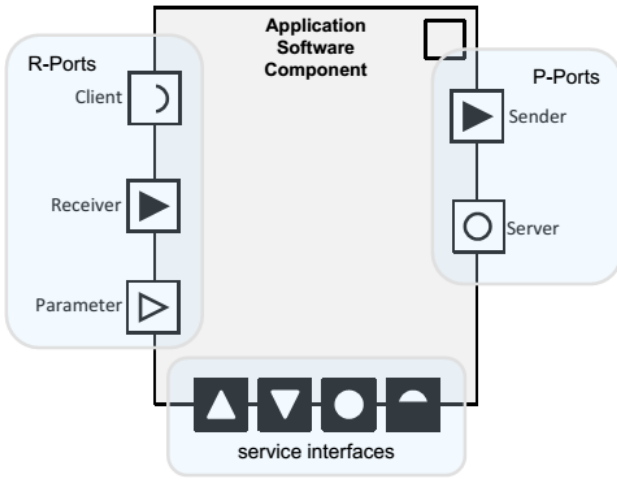


Figure 1.9: Software component ports, source:[GUL]

defined in AUTOSAR [AUT14a]:

- *Client-server interface.* The server provides operations that might be called by different clients.
- *Sender-receiver interface.* The sender sends data information to one or more clients that are consuming it.
- *Parameter interface.* One component can access constant or calibration data from other components.
- *Non-volatile interface.* One component has read/write access to non-volatile data.
- *Trigger interface.* Allows one component to trigger the execution of another software component.
- *Mode switch interface.* Used for notification of a software component of different states that the system can enter.

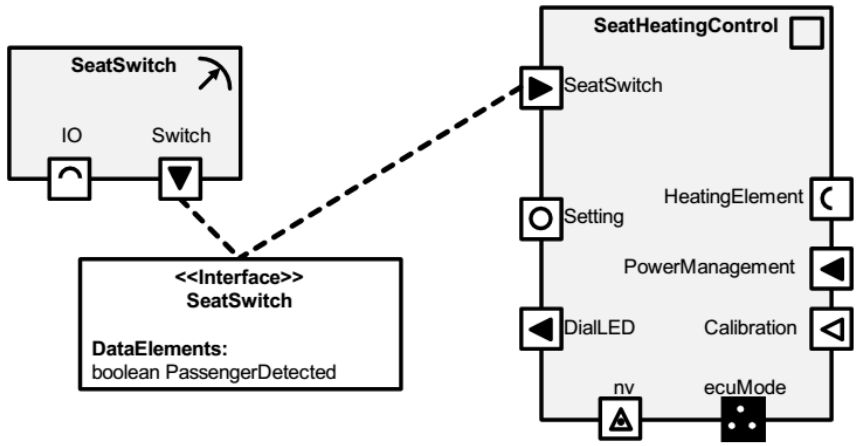


Figure 1.10: Software component port-inteface example, source:[AUT14a]

We will focus on *sender-receiver* interface in the thesis. The fig. 1.10 presents two components that communicate using *sender-receiver* interface. One component is producing a value for variable "PassengerDetected" and is sending the value using its *P-port* to the other component that is reading it through it's *R-port*. The listing 1.1 shows an implementation of a *software component* for a seat heating controller. Each *software component* is defined by its communication ports (lines 9-16) and its internal behavior (lines 20-35). The code has been written using Arctic Studio from ArcCore[6]. It is worth noting that an AUTOSAR *software component* can also be implemented in Matlab using Embedded Coder[7]. One can see that the main advantage of an AUTOSAR application is that the implementation does not dependent on any hardware or OS, therefore it can be reused on different ECU configurations.

Listing 1.1: Software component implementation in Arctic Studio

```
1  package Tutorial.HeatingController
2
3  import Tutorial.Interfaces.*
4  import HeatingControllerType.SetHeatingControllerBehaviour
5
6  component application HeatingControllerType{
7
8  ports{
9  receiver SeatSwitchLeft requires SeatSwitchStatusIf
10 receiver SeatSwitchRight requires SeatSwitchStatusIf
11
```

[6] http://www.arccore.com/
[7] http://se.mathworks.com/hardware-support/autosar.html

```
12 sender DialLedLeft provides DialLedIf
13 sender DialLedRight provides DialLedIf
14
15 client HeatingElementLeft requires HeatingElementIf
16 client HeatingElementRight requires HeatingElementIf
17 }
18 }
19
20 internalBehavior SetHeatingControllerBehaviour for
       HeatingControllerType {
21 runnable mainRunnable [1.0] {
22 serverCallPoint synchronous HeatingElementLeft.SetHeating
23 serverCallPoint synchronous HeatingElementRight.SetHeating
24 dataWriteAccess DialLedLeft.LEDStatus
25 dataWriteAccess DialLedRight.LEDStatus
26 timingEvent 2.0
27 }
28
29 runnable readDataRunnable [1.0] {
30 dataReceivedEvent  SeatSwitchLeft.SwitchStatus as
       DataLeftAvailableEvent
31 dataReceivedEvent  SeatSwitchRight.SwitchStatus as
       DataRightAvailableEvent
32 dataReadAccess SeatSwitchLeft.SwitchStatus
33 dataReadAccess SeatSwitchRight.SwitchStatus
34 timingEvent 4.0
35 }
36
37 }
38
39
40 implementation SetHeatingControllerApplication for
       SetHeatingControllerBehaviour {
41 language c
42 codeDescriptor "src"
43 }
```

1.3.2 Runnables (functional entities)

The behavior of a *software component* is constructed using the entities called *runnables*. They are software functions that implement the algorithms (behavior) of a *software component*. Each *runnable* has access to the port interfaces and can read/write data signals from/to other *software components*.

In AUTOSAR, each *runnable* execution must be triggered by an event. A *runnable* can start its execution when new data is available on its sender-receiver port (data receive event) or it can be triggered by a timer (timing event).

In listing 1.1 the behavior of the *software component* is implemented by two *runnables* called *mainRunnable* (line 21) and *readDataRunnable* (line 29). The *mainRunnable* has a write access to two sender-receiver ports called *DialLedRight* (line 24) and *DialLedLeft* (line 25) and its execution is triggered by a timing event (line 26). The *readDataRunnable* can be triggered by a timing event or by a data receive event when new data is available at one of its ports : *SeatSwitch-Left* (line 30) and *SeatSwitchRight* (line 31).

Once a *software component* is mapped to an ECU, the configuration and generation of the Runtime Environment (RTE) is applied. This implies that the necessary "glue code" will be created in order for the RTE layer to be able to trigger each *runnable* when an event such as new data is received or a timer expiration happens. An example of such RTE "glue code" generated for the heat controller *software component* is shown in listing 1.2. It can be seen that for both *runnables*: *mainRunnable* and *readDataRunnable*, a new wrapper function is generated and it has the format *Rte_<software component* name*>_<runnable name>*. It is important to emphasize that these two wrapper functions represent the "real *runnables*" that will be triggered by the RTE layer.

In the case of the *readDataRunable*, the RTE layer first gets the data from the sender-receiver port *Switch Status* (lines 26-37) and only after that it calls the *readDataRunable* of the *software component* (line 38).

Each *runnable* can only run in the context of an Operating System task. The task can also be seen as a container that provides the stack-space for a *runnable* to execute. More about tasks in the context of the AUTOSAR will be explained in the section 1.3.3.

Listing 1.2: Example of generated RTE "glue code"

```
1  /** === Runnables
       ================================================================
2  */
3  #define HeatingControllerType_START_SEC_CODE
4  #include <HeatingControllerType_MemMap.h>
5
6  /** ------ heatingController
       ------------------------------------------------
7  */
8  void Rte_heatingController_mainRunnable(void) {
9  /* PRE */
10
11 /* MAIN */
12
13 mainRunnable();
14
15 /* POST */
16 Rte_Write_HeatingControllerType_heatingController_
```

```
17 DialLedLeft_LEDStatus
18 (ImplDE_heatingController.mainRunnable.DialLedLeft.LEDStatus.value)
       ;
19
20 Rte_Write_HeatingControllerType_heatingController_
21 DialLedRight_LEDStatus
22 (ImplDE_heatingController.mainRunnable.DialLedRight.LEDStatus.value
       );
23 }
24
25 void Rte_heatingController_readDataRunnable(void) {
26 /* PRE */
27 Rte_Read_HeatingControllerType_heatingController_
28 SeatSwitchLeft_SwitchStatus(
29 &ImplDE_heatingController.readDataRunnable.
30 SeatSwitchLeft.SwitchStatus.value);
31
32 Rte_Read_HeatingControllerType_heatingController_
33 SeatSwitchRight_SwitchStatus(
34 &ImplDE_heatingController.readDataRunnable.
35 SeatSwitchRight.SwitchStatus.value);
36
37 /* MAIN */
38 readDataRunnable();
39 /* POST */
40
41 }
42 #define HeatingControllerType_STOP_SEC_CODE
43 #include <HeatingControllerType_MemMap.h>
```

1.3.3 OSEK Os and Schedulability

The AUTOSAR Operating System is based on the industry standard OSEK OS[8]. AUTOSAR has adopted a fixed priority fully preemptive scheduling policy. The unit of execution inside AUTOSAR OS is called an *Os-Task*. Each *Os-Task* has assigned a priority and it can always be preempted by another *Os-Task* with a higher priority value.

Two types of *Os-Task* are defined:

- Basic task fig. 1.11 that can be in one of the states: *ready* (the task waits for the allocation of the processor), *running* (the task has the processor and it executing its instructions), *suspended* (the task has released the processor to another task and can be re-activated).

[8] http://www.osek-vdx.org/

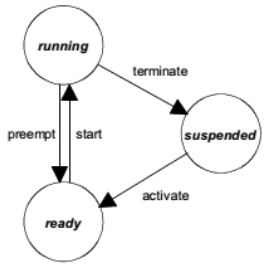


Figure 1.11: Basic Os-Task, source:[VDX05]

- Extended task fig. 1.12 that has one additional state to the ones of the basic task. In the *waiting state*, the task can blocked after calling a system service and can only be activated and put into *ready* state by an event such as timer expired or new data received.

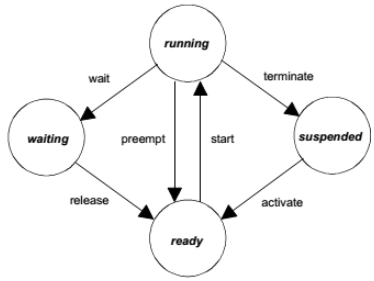


Figure 1.12: Extended Os-Task, source:[VDX05]

Each *runnable* from any *software component* needs to be mapped to an *Os-Task*. Multiple *runnables* can be assigned to the same *Os-Task*. According to [SR08], the simplest solution is to map each *runnable* into its own *Os-Task* but this is not feasible because the number of tasks can be limited in many systems and is not efficient (the core utilization overhead needs to be taken into account when the Operating System switches between tasks).

The fig. 1.13 and fig. 1.14 shows examples of *runnables* that use implicit or explicit communication mapped into a simple *simple and extended task*. If multiple *runnables* are mapped to a basic task, the condition is that they do not use API calls that might block the execution of the task. In this case, the RTE layer can trigger both *runnables* execution one after the other: fig. 1.13 a) and fig. 1.14 a).

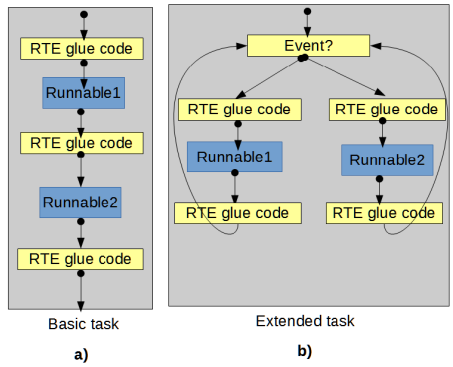


Figure 1.13: Runnables with implicit sender/receiver communication mode, source:[AUT14c]

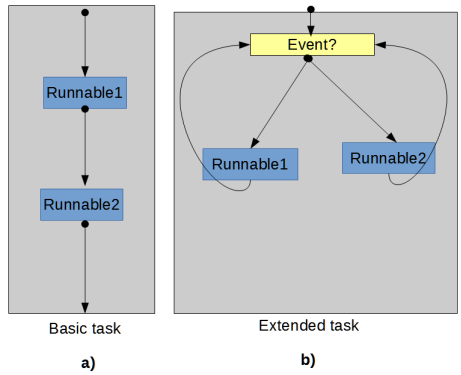


Figure 1.14: Runnables with explicit sender/receiver communication mode, source:[AUT14c]

If *runnables* that are mapped into the same task are triggered by different events such that one is time triggered every 10 ms and one every 20 ms, an extended task is used that waits in an endless loop for an event and then triggers the corresponding *runnable*: fig. 1.13 b) and fig. 1.14 b). An implementation example of an extended task that has three *runnables* mapped inside is presented in listing 1.3. The execution of the task is blocked at line 6, where it waits for an event to be triggered by the RTE layer. Based on the type of the triggering event, it calls the associated *runnables* (lines: 11,15,19,23,28).

Besides the rules defined by the AUTOSAR framework, other rules can be added depending on the system developer/integrator use cases.

For example one such rule could be that only *runnables* from the same *software component* may be allocated into the same *Os-Tasks*. An interesting study is presented in [LLP+09] where the authors had define new rules of mapping *runnables* to *Os-Tasks* such that they do not conflict with the AUTOSAR rules and the intra-ECU communication is minimized.

Regarding schedulability, in the case of multi-core ECUs, AUTOSAR specifies that each core is scheduled independently and a task from one core cannot preempt another task in the other cores.

Listing 1.3: Runnables mapped into an extended task

```
1  /** === Tasks
       ============================================================
2  */
3  void HeatingControllerTask(void) { /** @req SWS_Rte_02251 */
4          EventMaskType Event;
5          do {
6                  SYS_CALL_WaitEvent(
                       EVENT_MASK_DataLeftAvailableEvent |
                       EVENT_MASK_DataRightAvailableEvent |
                       EVENT_MASK_TimmingEvent | EVENT_MASK_OsEvent);
7                  SYS_CALL_GetEvent(TASK_ID_HeatingControllerTask, &
                       Event);
8
9                  if (Event & EVENT_MASK_TimmingEvent) {
10                         SYS_CALL_ClearEvent (
                               EVENT_MASK_TimmingEvent);
11                         Rte_heatingController_readDataRunnable();
12                 }
13                 if (Event & EVENT_MASK_DataLeftAvailableEvent) {
14                         SYS_CALL_ClearEvent (
                               EVENT_MASK_DataLeftAvailableEvent);
15                         Rte_heatingController_readDataRunnable();
16                 }
17                 if (Event & EVENT_MASK_DataRightAvailableEvent) {
18                         SYS_CALL_ClearEvent (
                               EVENT_MASK_DataRightAvailableEvent);
19                         Rte_heatingController_readDataRunnable();
20                 }
21                 if (Event & EVENT_MASK_TimmingEvent) {
22                         SYS_CALL_ClearEvent (
                               EVENT_MASK_TimmingEvent);
23                         Rte_heatingController_mainRunnable();
24                 }
25                 if (Event & EVENT_MASK_OsEvent) {
26                         SYS_CALL_ClearEvent (EVENT_MASK_OsEvent);
27                         Rte_ledLeft_DialMain();
28                 }
29
30         } while (RTE_EXTENDED_TASK_LOOP_CONDITION);
31 }
```

1.3.4 Os-Application

An *Os-Application* is an AUTOSAR entity that groups together a collection of Os-objects defined as *Os-Tasks*, Interrupt Service Routines, alarms, events, counters, etc. An *Os-Application* can be trusted, which means that each object that is part of it has unrestricted access to the API and hardware resources. Alternatively, each object of an untrusted *Os-Application* has limited access to the API and hardware resources and it runs in non-privileged mode. An example of an *Os-Application* declaration in ERIKA AUTOSAR[9] is given in listing 1.4.

Listing 1.4: ERIKA AUTOSAR *Os-Application* example, source: [AUTa]

```
1  APPLICATION App1 {
2  TRUSTED = FALSE;
3  TASK    = App1Task1;
4  TASK    = App1Task2;
5  TASK    = App1Task3;
6  COUNTER = App1Counter1;
7  COUNTER = App1Counter2;
8  ALARM   = App1Alarm1;
9  ALARM   = App1Alarm2;
10 SCHEDULETABLE = App1ScheduleTable;
11 SCHEDULETABLE = App1ScheduleTable;
12 //MEMORY_SIZE = 0x1000;
13 SHARED_STACK_SIZE = 256;
14 IRQ_STACK_SIZE = 256;
15 RESTARTTASK = App1Task3;
16 };
```

Each *Os-Application* has its own memory partition, separate stack, data and code. AUTOSAR assures that a code executed in the context of an *Os-Application* can not corrupt the memory area of another *Os-Application*.

[9] http://erika.tuxfamily.org/wiki/index.php?title=ERIKA_%26_Autosar_OS_Requirements

1.3.5 Communication

The main communication paradigms defined in AUTOSAR are the *sender-receiver*, *client-server* and *inter-runnable* paradigm.

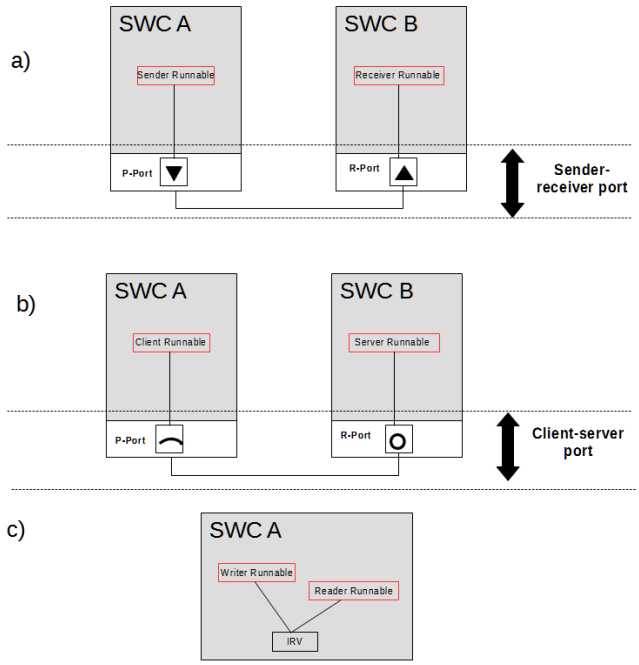


Figure 1.15: AUTOSAR communication paradigms

The *sender-receiver* paradigm fig. 1.15 a) defines an asynchronously communication mode where one sender *runnable* transmits data elements through its *component P-Port* and one or more receiver *runnables* are consuming the data through their *component R-Port*. The data can be of primitive type like "integer" or of complex type such arrays or structures. AUTOSAR defines two types of *sender-receiver* communication: *explicit sender-receiver* and *implicit sender-receiver*. Depending of the type of the *sender-receiver* chosen, the glue code generated in the RTE layer calls a different API.

The table 1.1 presents the signatures of the RTE API functions generated based on the type of the sender/receiver where the notation <re> means the *runnable* name, <p> means the *software component* port name used for sending the data and <d> means the name of the data that is send. Std_ReturnType is the return error code from the RTE API call.

Table 1.1: RTE API for sender/receiver mode, source:[Fre11]

RTE API function	Description
void Rte_IWrite_<re>_<p>_<d> ([IN RTE_Instance], IN <type>)	Implicit write with last is best semantic
<return> Rte_IRead_<re>_<p>_<d> ([IN Rte_Instance <instance>])	Implicit read with last is best semantic
Std_ReturnType Rte_Write_<p>_<d> ([IN Rte_Instance <instance>], IN <data>)	Explicit write with last is best semantic
Std_ReturnType Rte_Read_<p>_<d> ([IN Rte_Instance <instance>], OUT <data>)	Explicit read with last is best semantic
Std_ReturnType Rte_Send_<p>_<d> ([IN Rte_Instance <instance>], IN <data>)	Explicit write with queued semantic
Std_ReturnType Rte_Receive_<p>_<d> ([IN Rte_Instance <instance>], OUT <data>)	Explicit read with queued semantic

In the case of implicit *sender-receiver* interface, AUTOSAR defines the following types of behavior:

- The sender *runnable* can modify the data signal while is running but the RTE layer will only send the latest data of the signal after the *runnable* has finished the execution.

- The RTE layer generates a copy of the data signal before it triggers the receiver runnable. Before the execution of the receiver *runnable* starts, it reads the copy of the data signal and uses it during its entire execution time.

In the case of the explicit *sender-receiver* interface, AUTOSAR defines the following behaviors:

- The sender *runnable* can call the RTE API when it wants to send the data signal. Several calls to the API cause several transmissions of the data signal.

- The receiver *runnable* can call the RTE API when it wants to read the data. Several API calls to read the data signal results in different values retrieved.

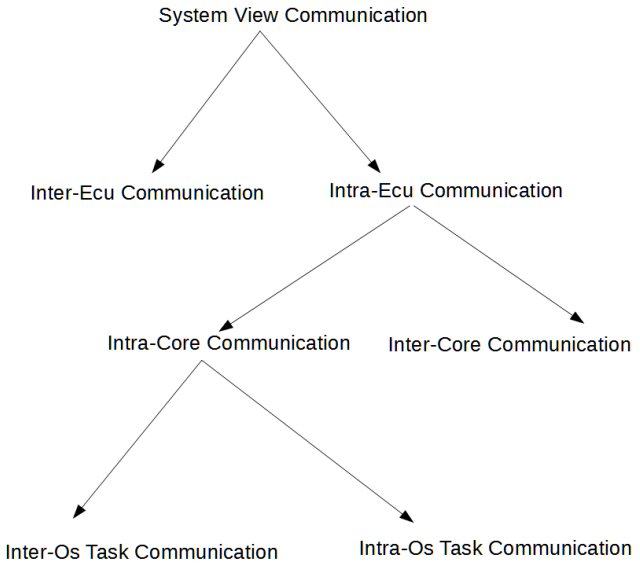


Figure 1.16: Types of communication at the system level

The *explicit sender-receiver* can be split further into queued communication and unqueued communication with last-is-best semantic. A buffer of fixed length is provided for the queued communication and the data signal is retrieved in FIFO (first-in first-out) order. In the case of unqueued communication with *last-is-best semantic*, the RTE layer assures that the *runnables* read the latest value of the data signal. An use case for using *sender-receiver with last-is-best semantic* is for example of a *runnable* which reads the value of a temperature sensor. In this case, it will need to get the latest value and using a queue is not recommended.

The *client-server* paradigm fig. 1.15 b) defines a bidirectional communication mode where one client *runnable* invokes the service of a server *runnable* that does some computation and return the result to the invoking client. The *runnable* might be blocking while waiting for the response (synchronous communication) or non-blocking (asynchronous communication). Many *runnable* clients can make an arbitrary number of requests to the server *runnable* who executes them in FIFO order.

The *inter-runnable* paradigm in fig. 1.15 c) can be threated as a special case of *sender-receiver*. *Runnables* within the same *software component* communicate asynchronously and they access the same inter-runnable variable for writing and reading a data value.

Based on the mapping of the *software components* to the ECUs and the *runnable's* mapping to cores and their grouping into *Os-Tasks*, we have classified in fig. 1.16 the types of communication that can occur at the system level.

1.3.5.1 Inter-ECU communication

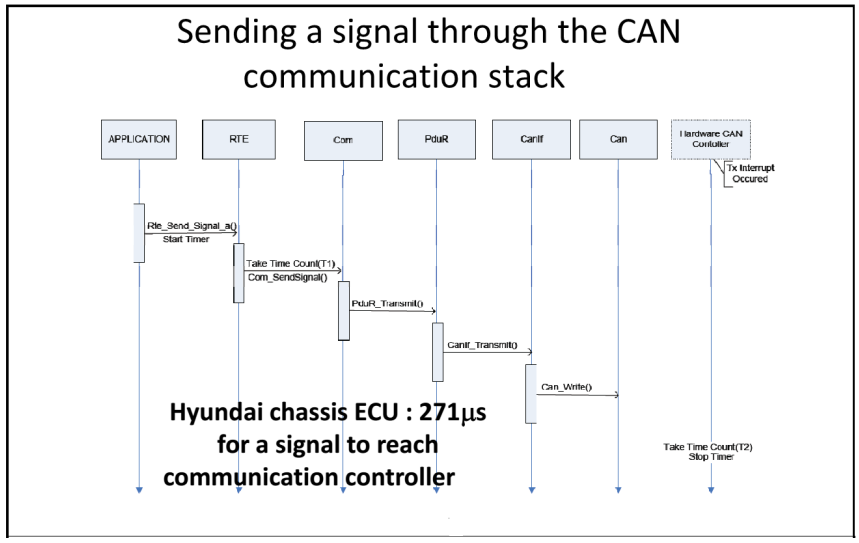


Figure 1.17: Inter-ECU communication over network bus [HC08]

The inter-ECU communication happens for *runnables* which are part of *software component* that are mapped to different ECUs when data signals are being send between these *runnables*. The sequence diagram in fig. 1.17 shows an application which is sending a signal over a CAN network. In this case, the RTE layer has to rely on modules such as Communication Stack (COM) that gets the data signals from the RTE and sends them over the physical network bus.

Listing 1.5 presents an example of the code generated for two *runnables* that use inter-ECU communication. Based on the RTE function calls (lines 2 and 13) it can be observed that the type of communication between *runnables* is *explicit sender-receiver with last-is-best semantic*. In this example, both RTE functions are using the Communication Stack (COM) API to send and receive a signal that is send between ECUs (lines 6 and 17).

Listing 1.5: Com functions called inside a RTE function

```
1         /** ------ FreqReq */
2         Std_ReturnType Rte_Read_Tester_TesterProto_FreqReq_freq(/*
              OUT*/UInt32 * value) {
3         Std_ReturnType status = RTE_E_OK;
4
5         /* --- Receiver (FreqReqfreqISig) @req SWS_Rte_04505, @req
              SWS_Rte_06023 */
6         status |= Com_ReceiveSignal(ComConf_ComSignal_FreqReqSig,
              value);
7
8         return status;
9         }
10
11        /** ------ FreqReqInd */
12
13        Std_ReturnType Rte_Write_Tester_TesterProto_FreqReqInd_freq
              (/*IN*/UInt32 value) {
14        Std_ReturnType retVal = RTE_E_OK;
15
16        /* --- Sender (FreqReqIndfreqISig) @req SWS_Rte_04505, @req
               SWS_Rte_06023 */
17        retVal |= Com_SendSignal(ComConf_ComSignal_FreqIndSig, &
              value);
18
19        return retVal;
20        }
```

1.3.5.2 Inter-Core communication

The inter-core communication happens when the *runnables* who are exchanging data signals are mapped to the same ECU but on different cores. The inter-core communicator (IOC) layer is responsible for getting the data signals from the RTE layer and passing them between cores. Depending on the *sender-receiver* mode using queued or unqueued semantic, the AUTOSAR framework will generate the functions for the sender side as in listing 1.6.

Listing 1.6: Generated IOC functions for the sending a signal

```
1 Std_ReturnType IocSend_<IocId>[_<SenderId>](
2 <Data> IN //for the explicit sender-receiver with queued semantics
3
4 Std_ReturnType IocWrite_<IocId>[_<SenderId>](
5 <Data> //for the explicit sender-receiver with last is best
      sematics
6 )
```

On the receiver side, the AUTOSAR framework will generate the functions as in listing 1.7.

Listing 1.7: Generated IOC functions for the reading a signal

```
1 Std_ReturnType IocReceive_<IocId>(
2 <Data> OUT  //for the explicit sender-receiver with queued
      semantics
3
4 Std_ReturnType IocRead_<IocId>(
5 <Data> OUT  //for the explicit sender-receiver last is best
      sematics
6 )
7 )
```

In fig. 1.18 *runnables* mapped to different ECU cores communicate between each other with the help of the RTE layer and IOC layer. It is important to notice that the IOC layer is also responsible for communication between *runnables* that belong to *Os-Tasks* assigned to different *Os-Applications*. In this case, the generated code for the IOC layer will have to handle the crossing of the memory protection boundaries of the *Os-Applications*.

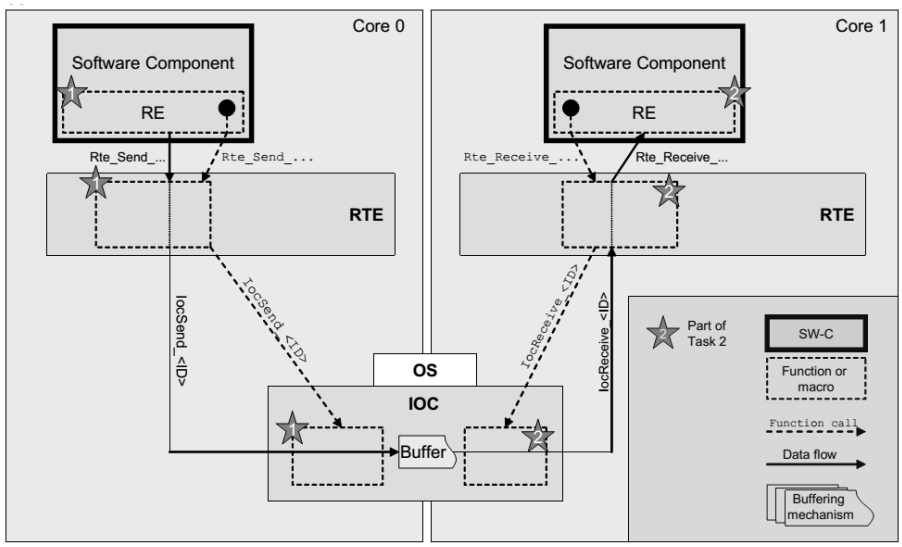


Figure 1.18: Runnable communication over IOC [AUT14b]

1.3.5.3 Intra-task and Inter-task communication

Intra-task communication happens when the *runnables* that are exchanging signals are mapped to the same *Os-Task*. The RTE layer is responsible for the transport of the data signals. In this case it is worth to mention that the order of mapping the runnables into the *Os-Task* is important. For example, the *runnable* that is sending the data should be executed inside the *Os-Task* before the *runnable* that reads the data.
Inter-task communication involves *runnables* that are mapped to different *Os-Tasks* on the same *core*. Again, the RTE layer is responsible for managing the exchange of data between the *runnables*.

1.3.6 Functional safety features

According to ISO 26262, if an automotive application contains *software components* with different ASIL levels, they have to be isolated such that a fault in the execution of one component does not affect the other ones. One approach is to implement all the *software components* according to the highest level of safety. This solution has an impact on the cost of the development and testing since more effort is needed such that all the *software components* are compliant (and probably needed to be certified) to the highest ASIL level.
Another approach is to use strategies than can assure freedom of interference for *software components* with higher ASIL from the ones with lower level. Techniques such temporal and spatial partitioning of *software components/runnables/Os-Tasks* can be used and are further explained in section 1.3.6.1 and section 1.3.6.2. It is worth to mention that in some cases the AUTOSAR Base Software modules, OS and RTE layer have to be implemented to support the highest level of ASIL as well. TTTech[10] already provides an AUTOSAR solution called MicrosarSafe[11] which is compliant with ASIL level D.

Possible faults might happen in AUTOSAR when:

- *A runnable* execution code can corrupt another *runnable/Os-Task's* memory area.
- *A runnable/Os-Task* exceedes its execution time.
- Communication corruption during *a runnable* exchanging data signals with other *runnables*.

When dealing with these types of faults, we apply as a solution spatial/temporal partitioning and end-to-end communication protection as they are proposed in the AUTOSAR.

[10] `https://www.tttech.com/`
[11] `https://www.tttech.com/products/automotive/autosar-safety-software/microsar-safe/`

1.3.6.1 Spatial partitioning

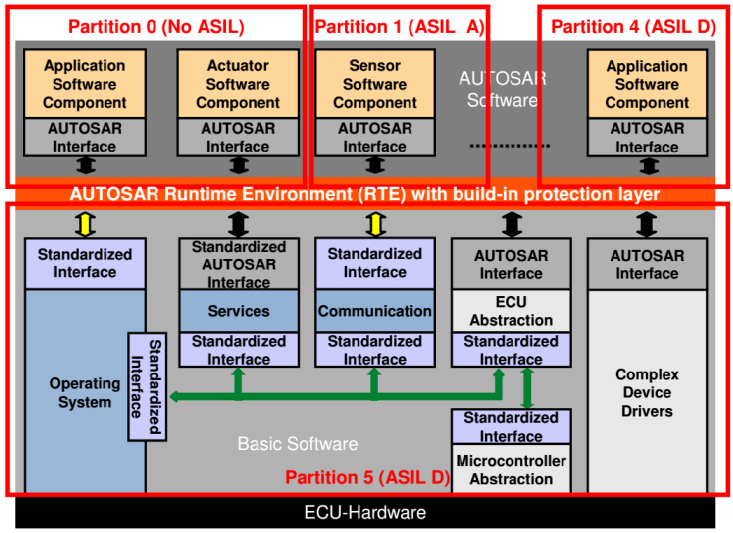


Figure 1.19: Memory partitioning example in AUTOSAR, source:[BFWS10]

Spatial or memory partitioning in AUTOSAR separates the *software components* such that neither of them can change/corrupt the memory associated with the other *software components*. Concretely, this is achieved with the help of *Os-Applications* (see section 1.3.4). For multi-core ECUs, AUTOSAR supports for each core the declaration of any number of *Os-Applications* that can be associated with one ASIL level.

An example of memory partitioning is presented in fig. 1.19. The modules from the Basic Software layer that is bellow the RTE layer usually have to run in their own partition. Furthermore, depending on the ASIL levels of each *software component*, they have to be grouped into different *Os-Applications*.

During run-time, if a fault or error appears in an *Os-Application*, the AUTOSAR OS can restart or stop it along with all the *Os-Tasks* associated. Therefore it might be necessary that an *Os-Application* to group *Os-Tasks* of the same ASIL, although it is not enforced. An AUTOSAR OS requires that the cores on the ECU contain MPU (memory protection unit) hardware to effectively support the separation of *Os-Application's* memory regions.

1.3.6.2 Temporal partitioning

Temporal partitioning is not enforced in AUTOSAR due to the fix priority preemptive scheduler. Based on how the priorities are assigned to the *Os-Tasks*, it is possible that an *Os-Task* that groups *runnables* of ASIL-D to be preempted by one with higher priority that has ASIL-C, therefore independence from the interference is not achieved.

The Operating System provides mechanisms against timing faults at the *Os-Task* level, not at a *runnable* level. A timing fault occurs when an *Os-Task* is missing its deadline at run-time. A *runnable* has to be mapped into its own *Os-Task* if the timing protection needs to be enabled only for it. The timing protection budget (TPB) for an *Os-Task* is enabled by configuring the following budget values:

- *Execution time budget.* The maximum amount of time an *Os-Task* is allowed to execute.
- *Resource lock time budget.* The maximum amount of time an *Os-Task* can hold a resource.
- *Inter-arrival time budget (Time Frame).* The minimum amount of time between successive activations of a basic *Os-Task*. In case of an extended *Os-Task* is the minimum amount of time between successive activations and releases.

The above budgets are configured statically (listing 1.8) and are used by the OS to check if an *Os-Task* has exceeded them at run-time. With this mechanism, the interference between *Os-Tasks* is bounded, therefore the *ISO 26262 absence of error propagation* requirement is met.

Listing 1.8: Timing budgets for an Os-Task, source: [AUTb]

```
1  TASK TaskPrio2 {
2   TIMING_PROTECTION = TRUE {
3   TIMEFRAME = 0.0025;  /* Two Activations of this TASK have to be
        separated by 250us to be accepted. (Provided that TASK
        activation number is respected) */
4   EXECUTIONBUDGET = 0.0005;
5   RESOURCE = RESOURCELOCK {
6                   RESOURCELOCKTIME = 0.0002;
7                   RESOURCE = RES_SCHEDULER;
8                 };
9         };
10 };
```

Figure 1.20 shows how OS starts the monitoring of the timing budgets when an *Os-Task* makes a transition from one state to another. In the case that one of the *Os-Tasks* exceeds its budget values, it can be killed or restarted by the OS along with all the *runnables* inside. The authors in [FFR12] proposed a

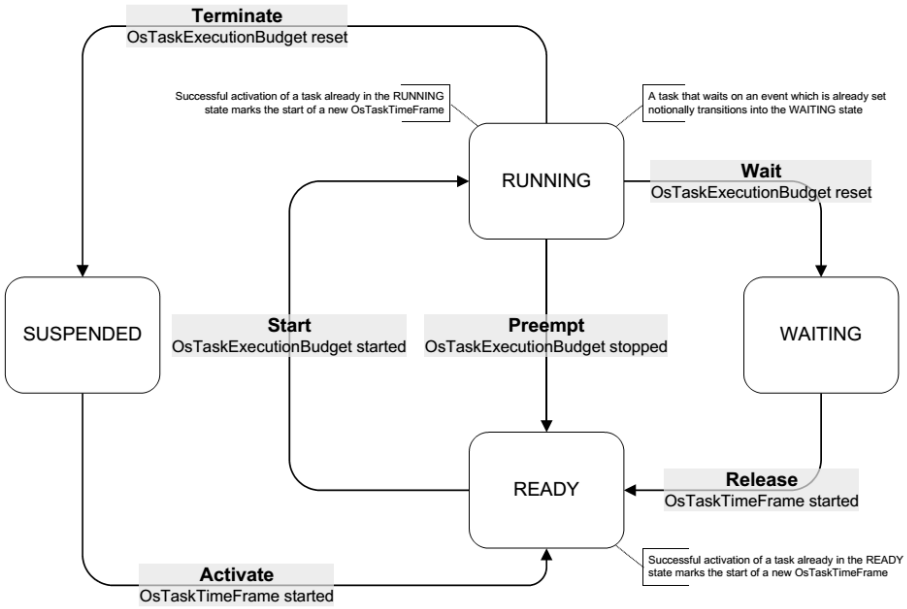


Figure 1.20: AUTOSAR OS timing monitoring, source:[AUT14b]

model for determining the timing protection budget of AUTOSAR *Os-Tasks* with different ASIL levels such that all of them meet their deadlines. They suggest that depending of the ASIL level of an *Os-Task*, the *execution time budget* should be computed as follows:

- ASIL-D: execution budget = WCET + 5%
- ASIL-C: execution budget = WCET + 10%
- ASIL-B: execution budget = WCET + 15%
- ASIL-A: execution budget = WCET + 20%
- ASIL-0: execution budget = WCET + 25%

The authors assume that for high critical *Os-Tasks*, the developers will spend more effort to get a good WCET compared to ones with a lower ASIL level, therefore the execution budget will be more pessimistic for *Os-Tasks* with lower ASIL.

It is also proposed that for an *Os-Task*, the WCET for *runnables* inside have to be determined and then the sum of all values obtained defines the WCET of the *Os-Task*. The authors further suggest that critical *runnables* should be placed in their own *Os-Tasks* to reduce the risk of being disturbed by others or if it's not possible, they should be executed first inside an *Os-Task*.

1.3.6.3 End-to-end communication protection

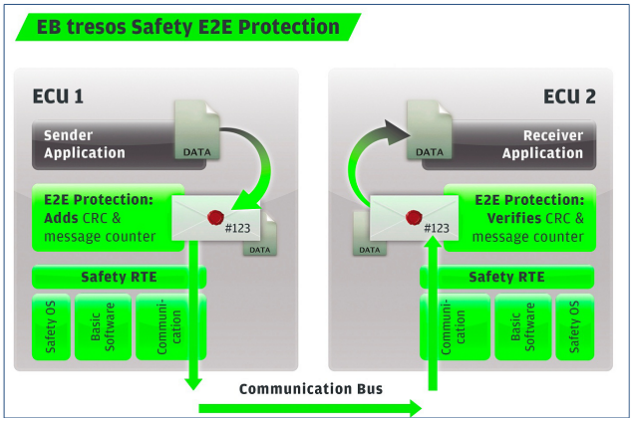


Figure 1.21: End-to-end protection example in AUTOSAR EB tresos product, source: [Mat14]

The end-to-end communication protection between *runnables* exchanging data signals is provided by the AUTOSAR E2E library. The library allows the sender *runnable* to transmit safety-related data and the receiver *runnable* to be able to handle and detect errors from the communication link. For example, a checksum number and a message counter can be assigned to each signal such that corrupted signals can be detected with cyclical redundancy check (CRC) while the missing signals can be identified based on their message counter fig. 1.21.

The fig. 1.22 presents possible sources of errors that can appear inside the AUTOSAR framework or at the hardware layer. E2E library has being designed such that it can handle faults generated by hardware failure or those produced at the RTE, IOC and COM layers.

In fig. 1.23 it can be observed that the E2E library is positioned between *runnables* exchanging data signals and the RTE layer. The main responsibility of the library is to wrap the safety-related information together with the real data from the sender and forward it to the RTE layer.

Next, the RTE layer will send the data via IOC layer since the *runnables* are part of different *Os-Applications*. On the receiver side, the *runnable* will make a request to E2E library to read the data. The E2E library will request the data from the RTE layer, unwrap the safety related information, check and detected any errors and then forward the actual data to the *runnable* to be consumed.

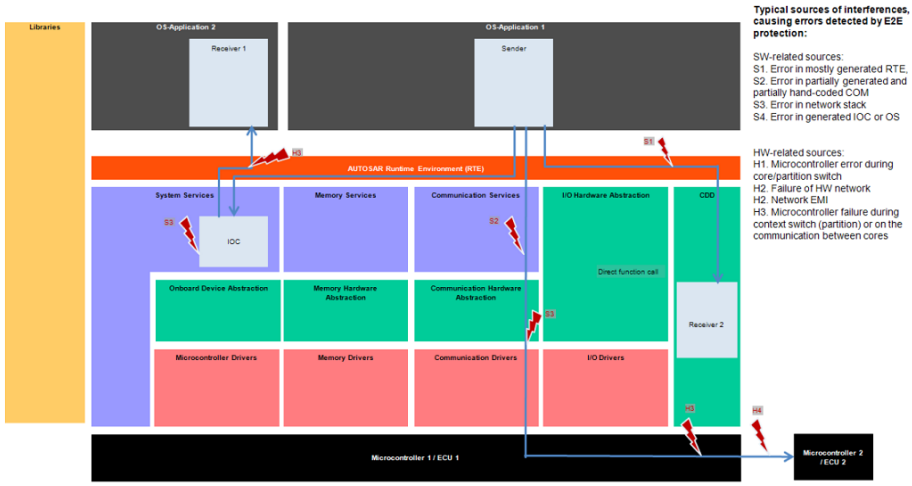


Figure 1.22: End-to-end possible faults, source:[AUT14d]

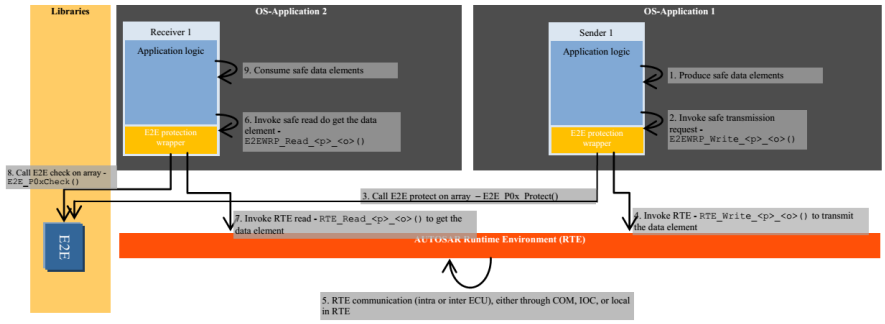


Figure 1.23: End-to-end communication protection, source:[AUT14d]

Chapter 2

System model

In this chapter, we present the application, architecture and AUTOSAR models for our mapping problem.

2.1 Application model

We have defined an AUTOSAR application as a set of *software components*:

$$Application = \{Software\ Component\} \tag{2.1}$$

Each *software component* contains a number of *runnables* (functions):

$$Software\ Component_i = \big(\{Runnable\}, ASIL\ level_i\big) \tag{2.2}$$

Every *software component* is assigned an ASIL level according to ISO 26262 functional safety standard. We define a *runnable* as:

$$Runnable_i = \Big(WCET_i, T_i, D_i, O_i, ASIL\ level_i, \{\big(R_j, signal\big)\}\Big) \tag{2.3}$$

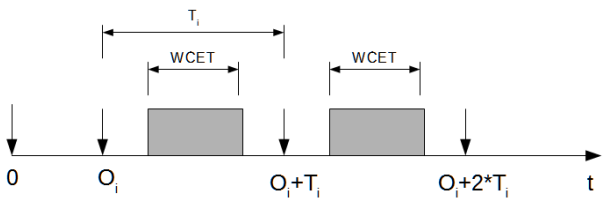


Figure 2.1: Runnable model

A *runnable* fig. 2.1 is characterized by the following attributes:

- $WCET_i$ - the worst-case time necessary for a *runnable* to execute its instructions without being interrupted.
- T_i - period of a *runnable*.
- $D_i = T_i$ - deadline of a *runnable*.
- O_i - offset of *runnable*.
- $ASIL\ level_i$ - *ASIL A, ASIL B, ASIL C, ASIL D.*
- $\{(R_j, signal)\}$ - a list of *runnables* that communicate and the amount of data in bytes send between them.

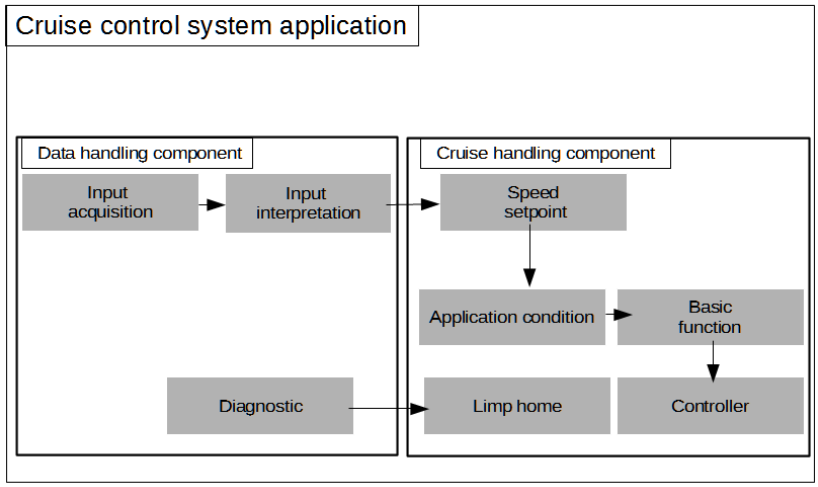


Figure 2.2: Control cruise application

Our model works with periodic *runnables* that are activated by an event timer and execute every T periods of time. Other assumptions for a *runnnable* are that the period is equal to its deadline and it starts executing its instructions right after being activated (offset = 0). Regarding ASIL level, each *runnable* will inherit the ASIL level of the *software component* it belongs to.

An example of an application model is presented in fig. 2.2. It has been adapted from the automotive use case described in the paper [ATPK+11].

The application implements the logic for a cruise control system in the vehicle and is composed of two *software components*. The first *software component: Data handling* contains three *runnables* responsible for the acquisition of data and diagnostics. The second *software component: Cruise handling* has five *runnables* controlling the vehicle speed in cruise mode. The arrows between *runnables* represent the exchanged data signals.

2.1.1 The WCET of a runnable entity

In this thesis, we define the WCET of a *runnable* as composed of the $WCET_{computational}$ which is the amount of time needed by a *runnable* to execute its instructions without interacting with the AUTOSAR RTE layer and $WCET_{communication}$ which is the amount of the time spent by a *runnable* when it is using the RTE layer's API for communication.

All the studied literature ignores the time it takes for a *runnable* to communicate over RTE and we consider that due to the complexity of the AUTOSAR framework and because of the safety related features introduced for communication, the values should not be ignored. Depending of how *runnables* communicate with each other, the RTE layer will use IOC layer (for inter-core communication), COM (communication stack) layer for inter-ECU communication or E2E library for reliable communication. Without splitting the WCET of a runnable, a system/software developer for AUTOSAR might have to measure the time spent for each *runnable* that is sending/receiving a data signal for inter-ECU, inter-core and inter-task communication and select the highest value as the WCET. Such strategy will result in a pessimistic WCET that will guarantee the safety and correct functionality of the system but it might allow a mapping tool to generate a solution that has high utilization of communication bandwidth between ECU cores or between ECUs. For example, due to high WCET, some *runnables* will have to be mapped into different cores even thought it would have been better to stay on the same core.

Because the development of a *software component* together with all its *runnables* is hardware independent, a system/software developer can take advantage of different timing analysis techniques such static code analysis and code simulation analysis to determine the WCET of all the *runnables*. On the other hand, one can also determine the WCET communication for inter-task, inter-core and inter-ECU communication by simulation or tracing techniques.

An interesting study is given in [GHAG11] by engineers from Gliwa GmbH, Infineon Technologies AG and SYMTAVISION GmbH where they proposed a strategy for migrating AUTOSAR application from single-core ECUs to multi-core. Using SymTA/S[1] timing analysis, they were able to measure the communication overhead for *runnables* exchanging signals. The values obtained were taken into account when computing the core utilization. The results in fig. 2.3 show that depending of how *runnables* exchanging signals are being mapped, the total communication overhead per core can be quite high: 20% for Core2 in **Core Load** chart.

Name	Buffer Access Overhead
R1	[76500 ns;112200 ns]
R2	[89675 ns;116025 ns]
R3	[70975 ns;127925 ns]
R4	[112625 ns;130000 ns]
R5	[48450 ns;61625 ns]
R6	[121200 ns;156000 ns]
R7	[42075 ns;49725 ns]

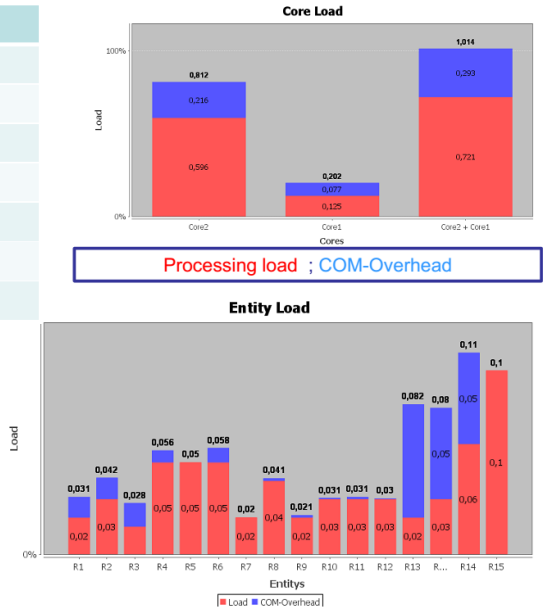


Figure 2.3: Communication overhead values, source:[GHAG11]

[1] http://www.symtavision.com/products/symtas-traceanalyzer/

2.2 Architecture model

The architecture consists of a hardware model running on top of an AUTOSAR solution.

2.2.1 Hardware architecture model

The hardware architecture consists of a number of heterogeneous multi-core and single-core ECUs that are connected through a shared network bus (e.g. CAN, FlexRay).

$$Architecture = \{ECU\}$$

Each ECU has a number of interconnected cores, an associated ASIL level and a *communication bandwidth (bytes/s)* defined for each pair of ECUs that exchange data over the network bus (2.4).

$$\begin{aligned} ECU_i &= \left(\{comm\}, \{Core\}, ASIL\ level_i\right) \\ comm &= \left(bandwidth, ECU_j\right) \\ i, j &\in [0, number\ of\ ECUs)\,, i \neq j \end{aligned} \tag{2.4}$$

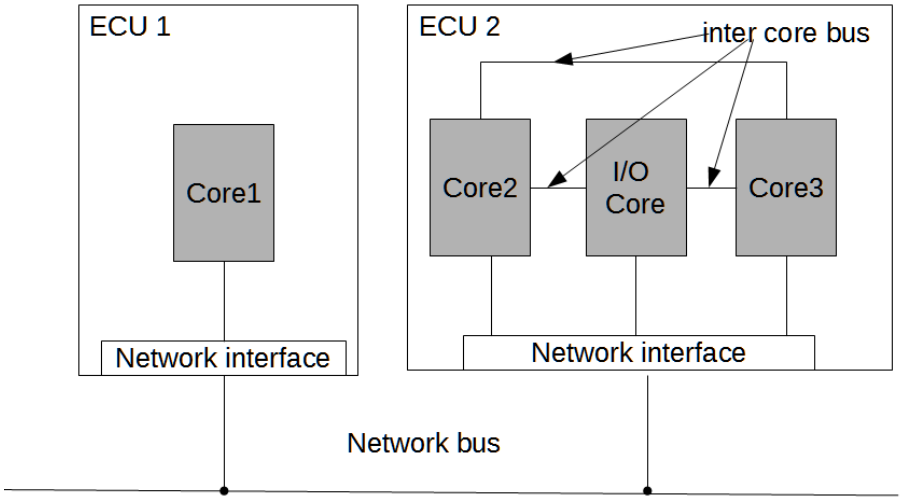


Figure 2.4: Hardware architecture model

Every core in the ECU has defined a *speed factor* and a *communication bandwidth* for each pair of cores that exchange data inside the ECU (2.5).

$$\begin{aligned} Core_i &= \left(\{comm\}, speed\ factor\right) \\ comm &= \left(bandwidth, Core_j\right) \\ i, j &\in [0, number\ of\ cores\ per\ ECU), i \neq j \end{aligned} \tag{2.5}$$

Since we are dealing with heterogeneous processing cores, a *speed factor* is defined as a value in $(0, 1]$ that represents how fast a *runnable's* instruction is executed on the current processing core compared to a reference one. For example, a speed factor of 0.5 suggests that an instruction will execute twice as fast compared to the reference core, therefore the WCET of a *runnable* will be reduced by half.

A possible use case is of an automotive application that has to be ported from a single-core ECU to a multi-core ECU. Since the WCET of all the *runnables* have already been determined for the single-core ECU, different speed factors can be used to see how the schedulability of the *runnables* mapped into multi-core ECU is affected.

An example of an hardware architecture is presented in fig. 2.4. There are two ECUs, one with a single core processor and one with a multi-core processor connected through a network bus. ECU1 models one single core architecture that has a CPU, memory, storage and peripherals. ECU2 is based on Freescale's Qorivva MPC5777M MCU[2]. It has two computational cores (Core2 and Core3) and one input/output core (I/O core) used for handling different hardware peripherals. It's worth noting that the ECU from Freescale is also ISO 26262 compliant up to ASIL level D.

2.3 AUTOSAR model

AUTOSAR software framework is running on each ECU in the hardware architecture model. A number of assumptions have been made regarding the scheduling, communication and safety in AUTOSAR.

2.3.1 Scheduling model

The schedulable entity in the AUTOSAR OSEK OS is an *Os-Task*. Every *runnable* from all the *software components* has to be mapped to an *Os-Task*.

$$Os - Task_i = (T_i, D_i, O_i, \{R\}, ASIL\ level_i)$$

[2] `http://cache.freescale.com/files/32bit/doc/fact_sheet/MPC5777MFS.pdf`

An $Os - task_i$ is characterized by the following attributes:

- $\{R\}$ - set of *runnables* mapped into task.
- $WCET_i$ - worst case execution time of a task. $WCET = \sum\{R.WCET\}$.
- T_i - period of a task. $T = \text{gcd}(\{R.T\})$.
- $D_i = T_i$ - deadline of a of a task.
- $ASIL\ level_i$ - *ASIL A, ASIL B, ASIL C, ASIL D.*

We have defined a single rule for mapping *runnables* into *Os-Tasks*: the *runnables* from the same *Os-Task* must have the same ASIL level.
Since we are working with periodic *runnables* and the periods of the *runnables* mapped inside a task might be different, we define the period of an *Os-Task* as equal to the greatest common divisor of all *runnables* inside. We assume that the WCET of an *Os-Task* is equal to the sum of all WCET of all *runnables* inside and that it's smaller than the period of an *Os-Task*.

According to AUTOSAR OS specification for multi-core ECUs [AUT14b], *Os-Tasks* are scheduled independently on each core. This means that the *runnables* inside each tasks are independent. Note that in our application model, there is a data dependency between *runnables* that communicate with each other, therefore our assumption is that the *runnable* that produces the data (sender) does not block waiting for an answer that the data signal has been received by the consumer *runnables*. On the other hand, for a *runnable* that consumes the data (receiver) we assume that by the time it starts executing, it already received all the data and is not waiting in a locked state.

Given this, we are using the schedulability test for independent tasks proposed by Liu & Wayland [LL73] with the difference that instead of computing the processing core utilization per task, we are computing the utilization per *runnable* mapped to the core. The core utilization per *runnable* is defined as the fraction of time spend on executing its instructions over its period. The utilization for each core is computed as the sum of all *runnable* utilization that are mapped to the core where $\boldsymbol{m}$ represents the number of *runnables* assigned to the processing core eq. (2.6).

$$\begin{aligned} U_{core} &= \sum_{i=1}^{m} \frac{R_i.WCET}{R_i.T} \\ R_i.WCET &- \ WCET\ of\ the\ runnable\ entity\ R_i \\ R_i.T &- \ Period\ of\ the\ runnable\ entity\ R_i \end{aligned} \tag{2.6}$$

According to [LL73] a sufficient condition for the tasks to be schedulable on a given core is that the utilization is below a specific bound given by $U_{core} \leq n \times (2^{\frac{1}{n}} - 1)$ where n represents the number of tasks (in our case *runnables*). A value given by the $\lim_{n\to\infty}(n \times (2^{\frac{1}{n}} - 1)) = \ln 2 \approx 0.69$ is usually used as the maximum core utilization value below which the tasks are schedulable.

The motivation for using the "runnable view" for computing the core utilization is that since not all *runnables* are activated when the task is running, the utilization might be over estimated in case we are using the "task view". As an example, if we have three *runnables* that are grouped into the same task and the *runnables* have the following properties: R1 (WCET = 5ms, T = 25ms) , R2 (WCET = 1ms, T = 50ms) , R3 (WCET = 10ms, T = 100ms), then the core utilization of the task is $WCET/T$ where $WCET = WCET_{R1} + WCET_{R2} + WCET_{R3}$ and $T = gcd(T_{R1}, T_{R2}, T_{R3})$. Therefore the utilization is : $U_{task\ view} = \frac{5+1+10}{gcd(25,50,100)} = \frac{16}{25}$. If we use the "runnable view", the utilization is only: $U_{runnable\ view} = \frac{5}{25} + \frac{1}{50} + \frac{10}{100} = \frac{8}{25}$.

2.3.2 The model of spatial partitioning

The main requirement of ISO 26262 for the integration of mixed critical systems is that safety relevant software must not be affected or disturbed. Spatial and temporal partitioning must be assured such that an error in a *runnable* with ASIL A level should not interference with a *runnable* of ASIL B level.

Spatial partitioning in the AUTOSAR framework is achieved through *Os-Application*. In order to isolate *runnables* with different ASIL levels, we only allow *runnables* from the same ASIL level to be mapped together into an *Os-Task*. Furthermore, for each core we have defined an *Os-Application* that will contain only tasks with the same ASIL level.

$$Os - Application_i = \big(ASIL\ level_i, \{OS\ task\}\big)$$

Each *Os-Application* will reside/use a different memory area (partitions) and AUTOSAR OS will assure that a *runnable* executing in one *Os-Application* can not modify a memory region from another *Os-Application*.

2.3.3 Communication model

Once the *software components* are mapped to each ECU and all the *runnables* are mapped to each core, the communication between *runnables* that exchange data signals is done through the AUTOSAR RTE layer. An example of *runnable*s communicating between ECUs and inside ECU is presented in fig. 2.5. *Runnables* from all the *software components* are communicating through sender-receiver ports and at the AUTOSAR Infrastructure layer it can be observed that the RTE is relying on the Basic Software (BSW) to send data between ECUs.

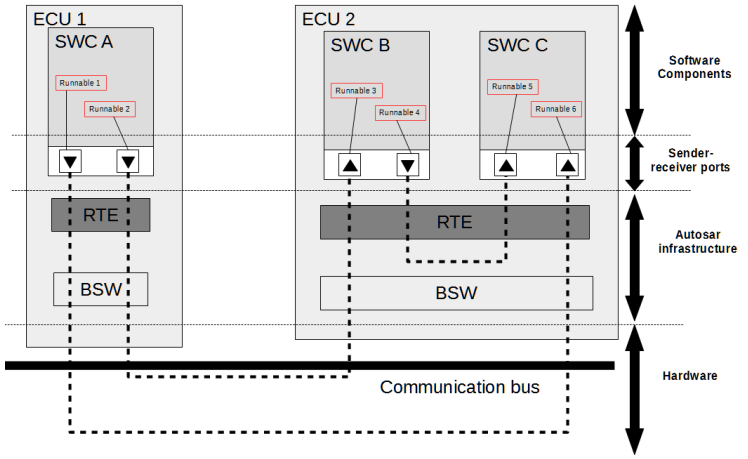


Figure 2.5: Runnable communication over AUTOSAR Runtime Environment (RTE)

In general, we can distinguish the following types of communication:

- Intra-ECU communication - between *runnables* exchanging data signals mapped on the same ECU.
  - Intra-Core communication - between *runnables* exchanging data signals mapped on the same core.
  - Inter-Core communication - between *runnables* exchanging data signals mapped on different cores.
- Inter-ECU communication - between *runnables* exchanging data signals mapped on different ECUs.

We define the *runnables* bandwidth for each core as the sum of the data signals exchanged by the *runnables* divided over their period eq. (2.7).

$$runnables\ bandwidth\ on\ ECU/core = \sum_{Runnable\ mapped\ into\ ECU/core} \frac{Runnable.signal}{Runnable.T} \tag{2.7}$$

The bandwidth utilization between *ECUs/cores* is the sum of all *runnable* bandwidth on *ECU/core* divided per maximum bandwidth of the *ECU/core* communication bus eq. (2.8). The values are between $[0, 1]$ with 1 meaning that the utilization of the communication bus is 100%.

$$bandwidth\ utilization = \frac{runnables\ bandwith\ on\ ECU/core}{maximum\ bandwidth\ of\ the\ ECU/core} \tag{2.8}$$

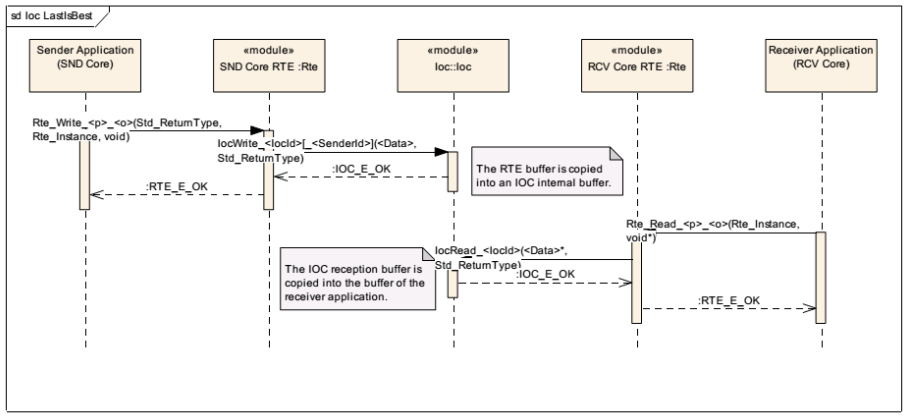


Figure 2.6: Sender-receiver with last-is-best model [AUT14b]

Our assumption for *runnables* exchanging data signals is that they are using asynchronous communication with non-blocking read/write operations. Therefore, the AUTOSAR *sender-receiver with last-is-best* mode of communication can be used for sending signals between *runnables*.

The sequence diagram in fig. 2.6 shows how *sender-receiver* mode works in the case of two *runnables* that are exchanging signals over RTE and IOC. It can be observed that the sender *runnable* does not wait until the signal is delivered and the receiver does not block until the signal becomes available which is in accordance with our assumption about asynchronous non-blocking read/write.

Based on the type of communication that we have defined and based on the AUTOSAR *sender-receiver* mode being used, we can define the $WCET_{communication}$ overhead for *runnables* exchanging data signals as:

- α = if *runnables* are mapped into the same *Os-Task*.
- β_0 = if *runnables* have the same ASIL levels and are mapped into different *Os-Tasks*.
- β_1 = if *runnables* have different ASIL levels and are mapped into different *Os-Tasks*.
- γ = if the *runnables* are mapped into *Os-Tasks* on different cores on the same ECU.
- θ = if the *runnables* are mapped into *Os-Tasks* on different ECUs.

We assume that $\theta > \gamma > \beta_1 > \beta_0 > \alpha$ since there is more work to be done by the AUTOSAR framework when *runnables* are mapped into different ECUs and/or cores. It is worth noting that the use of E2E library for reliable end-to-end communication will have also an impact on the $WCET_{communication}$ for each *runnables* exchanging data signals (see section 1.3.6.3).

Chapter 3

Functionality assignment to multi-core and optimization

This chapter presents the mapping algorithm proposed and the use cases for testing.

3.1 Problem formulation

Given an application model in section 2.1 and an architecture model as defined in section 2.2 we want to determine:

- A mapping of *software components* to ECUs.
- A mapping of *runnables* to cores.
- A mapping of *runnables* to *Os-Tasks*.
- A mapping of *Os-Tasks* to *Os-Applications*.

Such that we want to minimize:

O.1 The overall communication bandwidth. *Runnables* exchanging data signals should be mapped as closed as possible to each other (e.g into same *Os-Task* or into same *processing core* or into the same ECU).

O.2 The variance of the core utilization of the system. If we have an ECU with three computational cores, we want that the *runnables* are mapped such that one core does not have an utilization of 80% and the rest of the cores have 10%. We compute the variance of the *core utilization* to measure how far the values are from the overall mean.

Taking into consideration that:

C.1 The mapping constraints from AUTOSAR and the software developer/integrator are satisfied. For exemple, an AUTOSAR constrain specifies that *runnables* from the same *software component* have to be mapped on the same ECU.

C.2 The *runnables* are schedulable. *Runnables* have to be mapped such that for each procesing core the utilization is not greater than 69% so the schedulability test is met, see section 2.3.1 .

C.3 The *runnables* with different safety integrity levels are spatially and temporally isolated.

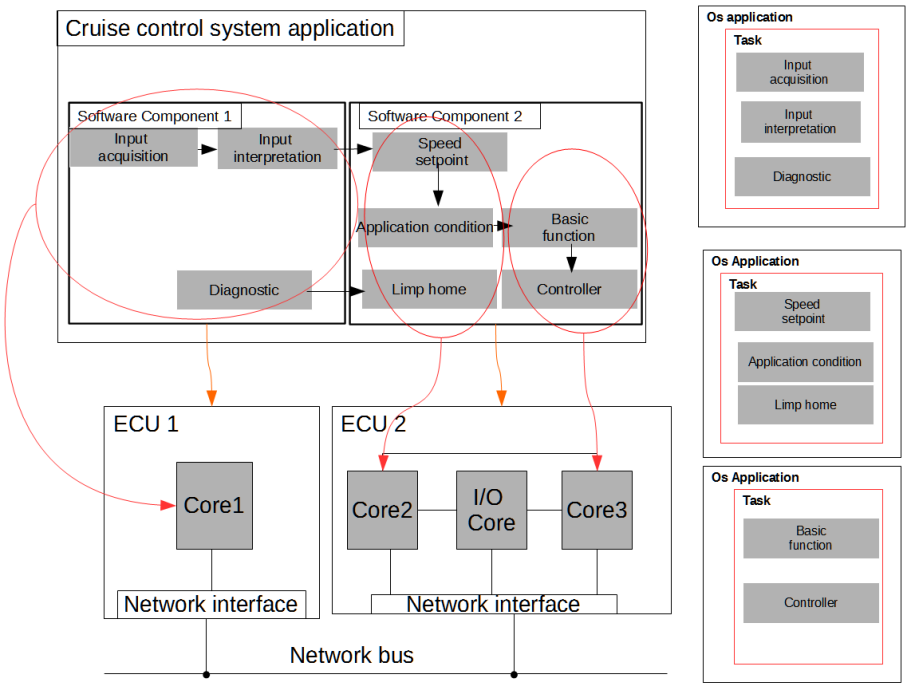


Figure 3.1: Mapping solution to ECUs

An example of mapping solution is given in fig. 3.1 where as an application model we have the one defined in fig. 2.2 and as an architecture model, we have the one defined in section 2.2.

One possible solution is to map the *software component 1* to ECU1 and the *software component 2* to ECU2. Next, all *runnables* from the *software component 1* will be mapped to Core1 and a possible mapping for *runnables* from *software component 2* to cores is:

- Speed Setpoint, Application condition and Limp home mapped to Core2.
- Basic function, Controller mapped to Core3.

After all the *runnables* are mapped to the cores, they must be grouped into *Os-Tasks*. As mentioned in section 2.3.1, runnables with the same *ASIL level* will be grouped into the same task on each processing core.
A possible grouping of *runnables* to *Os-Task* is:

- Core1 - one *Os-Task* with Input aquisiton, Input interpretation and Diagnostic
- Core2 - one *Os-Task* with Speed setpoint, Application condition and Limp home.
- Core3 - one *Os-Task* with Basic function and Controller.

The last step is to group the *Os-Tasks* into *Os-Applications* for each core. As explained in section 2.3.2, *Os-Tasks* with the same ASIL level will be grouped into the same *Os-Application* to isolate *runnables* with different ASIL levels from each other.

3.2 The solution space of the problem

Any of the mapping that we want to construct (*software components to ECUs, runnables to processing cores, runnables to Os-Tasks*) can be reduced to a bin-packing problem. If we have $\boldsymbol{n}$ elements such as *software components,runnables* and $\boldsymbol{k}$ bins as *ECUs, processing cores, Os-Tasks* we want to determine a solution composed of $\boldsymbol{k}$ disjoint subsets of elements (*software components, runnables*) that has the minimum variance in core utilization **O.2** and the minimum overall communication bandwidth utilization **O.1**, such that the constraints in **C.1**, **C.2**, **C.3** are met. The number of possibilities to generate $\boldsymbol{k}$ non-empty, disjoint subsets of $\boldsymbol{n}$ elements is given by the Stirling number of the second kind: eq. (3.1).

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{(k-i)}\binom{k}{i}i^n \tag{3.1}$$

The eq. (3.1) does not make any distinctions between the $\boldsymbol{k}$ bins, which is not the case in our problem, therefore we also have to add the cases when the set of all elements is assigned to one of the $\boldsymbol{k}$ bins.

Let's assume that we have $\boldsymbol{m}$ number of *software components* with $\boldsymbol{n}$ *runnables* each and $\boldsymbol{k}$ *ECUs* with $\boldsymbol{l}$ *processing cores* each. The number of ways we can map $\boldsymbol{m}$ *software components* to $\boldsymbol{k}$ *ECUs* is:

$$k! \times S(m,k) + k \tag{3.2}$$

where S(m,k) is obtained from 3.1

For each *software component* to *ECU* mapping, the number of ways to map $\boldsymbol{n}$ runnables to $\boldsymbol{l}$ processing *cores* is

$$l! \times S(n,l) + l \tag{3.3}$$

where S(n,l) is obtained from 3.1

Once we have all the *software components* mapped to the *ECUs* and all the *runnables* mapped to the *processing cores*, a grouping of *runnables* into *Os-Tasks* must be done for each processing core. Let's assume that the *runnables* are evenly distributed among all the *cores* with $\boldsymbol{r}$ such that

$$r = \frac{m \times n}{k \times l}$$

On each core, we can have all the *runnables* mapped into one *Os-Task*, or two *Os-Tasks* and so forth up to the case when each *runnable* is mapped to its own *Os-Task* eq. (3.4).

$$\sum_{i=1}^{i=r} S(r,i) \tag{3.4}$$

where S(r,i) is obtained from 3.1

To conclude, the number of possible solutions for our problem can be obtained by multiplying all the formulas from 3.2, 3.3, 3.4.

$$(k! \times S(m,k) + k) \times (l! \times S(n,l) + l) \times (\sum_{i=1}^{i=r} S(r,i)) \tag{3.5}$$

Given the case where we have one *software component* with thirty *runnables* and one ECU with *two processing cores*, the number of possible solutions are over one billion.

3.3 Cost function

A cost function has been defined eq. (3.6) to compute the variance of the core utilization **O.2** and the overall communication bandwidth **O.1** for a possible candidate from the solution space.

$$
\begin{aligned}
cost\ function = {} & W_1 \times (\sigma) \\
& + W_2 \times \left(\sum_{comm \in \{\cup\{ECU.comm\} \cup \{\cup\{ECU.\{Core.comm\}\}\}} comm \right) \\
& + penalty\ factor \times \left(\sum_{core \in \{\cup\{ECU.\{core\}\}\}} \max\left(0, U_{core} - U_{core\ max}\right) \right) \\
& + penalty\ factor \times \left(\sum_{comm \in \{\cup\{ECU.comm\} \cup \{\cup\{ECU.\{Core.comm\}\}\}} \right) \\
& max\left(0, U_{comm} - 1\right). \\
\sigma = {} & \frac{1}{N-1} \times \left(\sum_{core \in \{\cup\{ECU.\{core\}\}\}} \left(U_{core} - \mu\right)^2 \right) \\
\mu = {} & \frac{1}{N} \times \left(\sum_{core \in \{\cup\{ECU.\{core\}\}\}} U_{core} \right) \\
& Where\ N\ represents\ the\ number\ of\ cores\ in\ the\ architecture\ model.
\end{aligned}
\tag{3.6}
$$

The eq. (3.6) can be divided in three parts. The first part computes the variance of the core utilization weighted by W_1. We want that the mapping of *runnables* should be done in such a way that each core utilization is closer to the mean utilization.

The second part of the formula computes the sum of bandwidth utilization for each inter-ecu and inter-core communication link (see 2.8), weighted by W_2. We want that the *runnables* that are exchanging data signals to be mapped as close as possible to each other such that the overall inter-ECU and inter-core communication is minimized. For example, in the case of two *runnables* exchanging data signals, if they are mapped into the same core, no inter-ECU or inter-core bandwidth will be used.

The third part of the formula adds a penalty factor when either one of the core utilization is higher than a threshold given by a system/software developer or one of the communication bandwidth utilization is higher than 100%. For the experiments the threshold for core utilization factor was set to 0.69 (see 2.3.1).

3.4 Optimal solution

A backtracking algorithm has been implemented in order to explore all the solution space and select the optimal one. By optimal solution, we mean the one with the smallest cost (see eq. (3.6)). As explained in [Ski08], for the entire solution space of the problem, backtracking works by constructing a tree of partial solutions where each vertex represents a part of the final solution. In our case, a final solution means a mapping of *software components* to *ECUs*, *runnables* to *cores*, *runnables* to *Os-Tasks* and *Os-Tasks* to *Os-Applications*.

A backtracking algorithm works if the problem space is relative small. For our case the problem space is quite big as explained in the section 3.2. For an automotive application where we can have hundreds of *software components* with hundreds of *runnables* each, running on an hardware architecture with tens or even hundreds of ECUs, finding the optimal solution by exploring all the problem space can take an unfeasible amount of time and program memory space. Since the backtracking algorithm will always find the optimal solution given enough time and memory, for small application models it can be used to check how close to the optimal solution we can get with the metaheuristic algorithm introduced in section 3.5.

3.5 Simulated annealing

Simulated annealing is a heuristic search method for combinatorial problems. The method has been inspired from the process of heating and cooling metal materials to alter their physical properties. The process of physical annealing starts by heating the metal material over the melting point. While the material is cooling, the energy state of the system is changing. The transition probability from one energy level to another, at a given temperature, is given by eq. (3.7) :

$$P(\delta E, T) = e^{\delta E / k_B \times T},\ \delta E = E_{old} - E_{new} \tag{3.7}$$

where k_B is known as Boltzmann constant

According to the eq. (3.7), the probability of moving from a higher energy state to a lower energy state is high. Furthermore, there is still a non-zero probability to move to a higher energy state than the current energy state. The higher the temperature is, the higher is the probability of energy jumps. As it cools down, a physical system reaches a minimum energy state and the energy jumps are less likely to appear.

Table 3.1: Physical annealing and Simulated annealing, source:[Ree93]

Thermodynamic Simulation	Combinatorial Optimization
System States	Feasible Solutions
Energy	Cost
Change of State	Neighboring Solutions
Temperature	Control Parameter
Frozen state	Heuristic Solution

Given this, we can relate the process of physical annealing to finding the optimal solution is a combinatorial optimization problem. The table 3.1 makes the connection between the simulation of the physical annealing and exploring the solution space in a combinatorial optimization problem.

The solution space of the problem is equivalent to the thermodynamic states and the cost value of eq. (3.6) is the same as the energy of the system at a given temperature. When the system is changing the energy, this is identical for combinatorial optimization problem to a move from the current solution to a neighboring one. Simulated annealing is not more than an algorithm that mimics the physical annealing process. While exploring the solution space, it occasionally allows for jumps from a current solution to an inferior one to avoid getting stuck in a local minimum. A solution that has a minimum cost compared to the neighboring ones (local minimum) is not necessary the best (optimal).

The algorithm 3.1, takes as input an initial temperature (from where it starts to "cool down"), a minimum temperature where it stops to cool and the amount of steps done before lowering the temperature. The algorithm starts by computing an initial solution and the associated cost, lines 1, 2. For each temperature, a new solution is generated by applying a random transformation and computing a cost, lines 5, 6, 7. If the new cost is better than the old one, the new solution is accepted, otherwise the new solution is accepted with the probability given by a formula similar to 3.7, lines 9, 10, 11. After the amount of steps per temperature has been exceeded, the current temperature is "cooled down" by a cooling factor.

Input:
initial temperature
minimum temperature
max steps per temperature
cooling factor
Output:
Heuristic solution

```
1  Make an initial solution S;
2  Compute current cost;
3  while current temperature > minimum temperature do
4  |  for step := 1..max steps per temperature do
5  |  |  Randomly choose a move strategy;
6  |  |  Generate new solution S' by applying the move to the current
   |  |  solution S;
7  |  |  Compute new cost of S';
8  |  |  if new cost > curent cost then
9  |  |  |  current solution = new solution
10 |  |  else
   |  |  |     current solution = new solution with probability
11 |  |  |  α = e^((new cost−old cost)/current temperature)
12 |  |  end
13 |  end
14 |  current temperature = current temperature ∗ cooling factor
15 end
```

Algorithm 3.1: Simulated Annealing

3.6 Functionality mapping tool

A functionality mapping tool has been implemented. At the core of the tool stands a modified simulated annealing algorithm that takes as input an application model, an architecture model, any system mapping constraints and outputs a mapping of *software components* to ECUs, a mapping of *runnables* to cores, a mapping of *runnables* to *Os-Tasks* and a mapping of *Os-Tasks* to *Os-Applications* fig. 3.2. For our implementation of simulated annealing, an

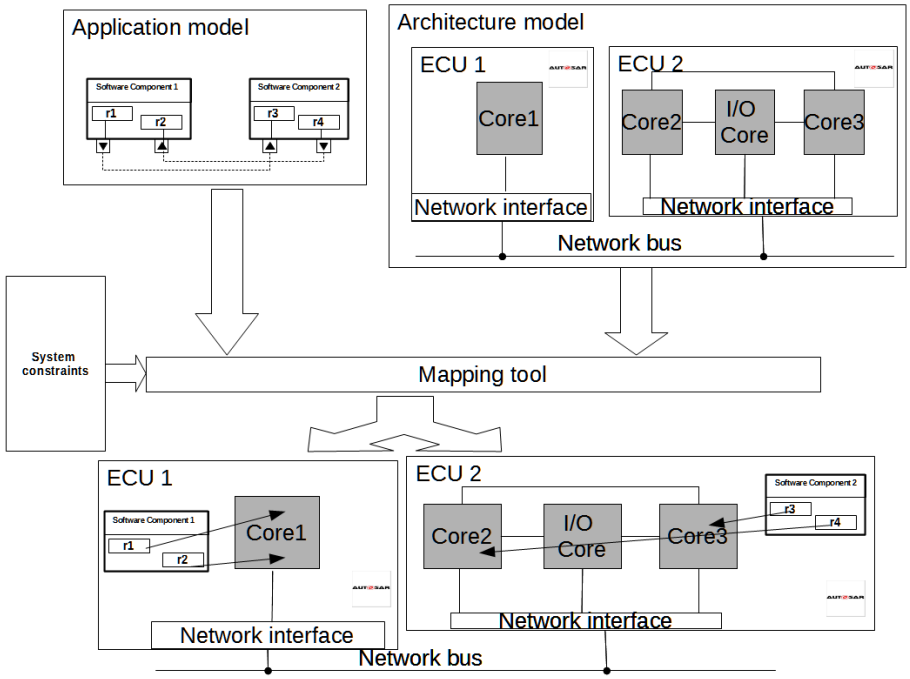


Figure 3.2: Mapping tool

exponential decay has been chosen for the temperature: $T_{new} = \alpha \times T_{old}$, where α is the cooling factor.

A solution represents a mapping of *software components* to ECUs, *runnables* to cores, *runnables* to *Os-Tasks* and *Os-Tasks* to *Os-Applications*. Consider the pseudo-code presented in algorithm 3.2. First, an initial solution is constructed (lines 1 - 7) by mapping each *software component* to a random ECU. Then each *runnable* is mapped to a random core of the ECU where the *software component* to which it belongs has already being mapped. After this, the cost of the mapping is computed based on the eq. (3.6).

For each temperature, we repeat the steps 8 - 21, before the lowering of the temperature starts.

We have defined three generation strategies that we want to apply to a given solution. At each step, one of the strategies is chosen randomly and applied to the current solution. For our problem, the strategies involve the following:

S.1 (fig. 3.3) Randomly choose a *software component* and map it to a new ECU which is different from the one where it is currently assigned. First, the ECU is selected at random. Second, all the *runnables* inside the *software component* are randomly mapped to the cores of the new ECU.

S.2 (fig. 3.4) Randomly choose a *runnable* and map it to a new core. The *core* is selected at random from the ones of the ECU where the *software component* that contains the runnable is mapped.

S.3 (fig. 3.5) Randomly choose two *runnables* that reside on the same core and group them together into an *Os-Task*. The only rule for grouping two *runnables* is that they must have the same ASIL level. It is possible to add new rules such that only *runnables* with the same period might be grouped together. At the beginning of the algorithm, all the *runnables* are mapped into their own *Os-Task*.

After a strategy has been chosen, it is applied to the current solution. Based on the type of the strategy selected, a new mapping of *software components* to ECUs, *runnables* to cores, *runnables* to *Os-Tasks* and *Os-Tasks* to *Os-Applications* is generated.

Given the new solution, the cost is computed based on the formula 3.6. It is wort to notice that based on how the *runnables* are being mapped, this will affect their $WCET_{communication}$ as explained in section 2.3.3 and the overhead is added to the utilization of the core where they are placed. Furthermore, the mapping of the *runnables* affects the inter-ecu and inter-core bandwidth utilization (see section 2.3.3) and the values are added to the cost for the current solution.
If the new cost is smaller than the cost before the transformation, the new solution is selected, lines 13, 14, otherwise we keep the current solution. If the probability of accepting the new solution is greater than a random generated number between $[0, 1)$, then the new solution is also selected, lines 16 - 20.

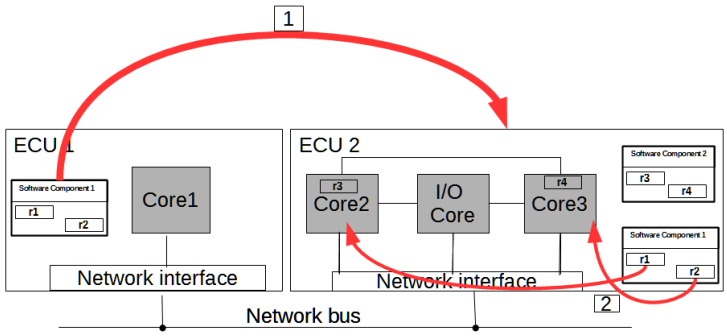


Figure 3.3: Move software component transformation

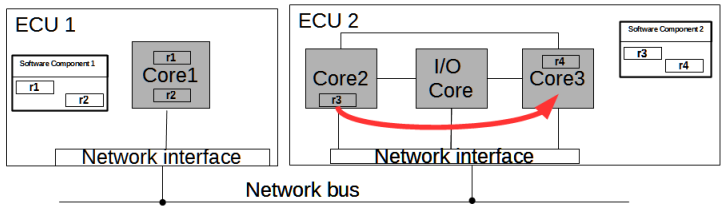


Figure 3.4: Move runnables between cores

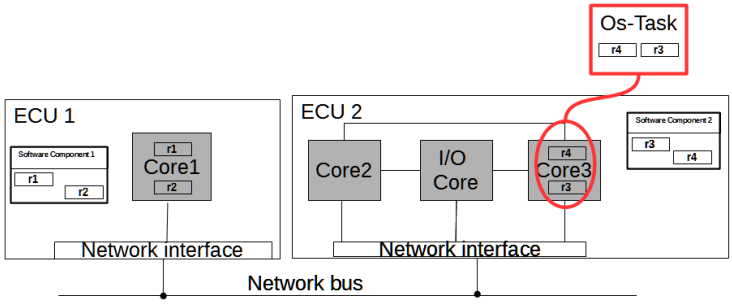


Figure 3.5: Move Runnables into same Os-Task

Input:
application model
architecture model
system mapping constraints
current temperature,minimum temperature,max steps per temperature
Output:
A mapping of software components to ECUs.
A mapping of runnables to Os-Tasks.
A mapping of Os-Tasks to cores.
A mapping of Os-Tasks to Os-Applications.

```
1  foreach software component in the application model do
2  |  randomly assign it to an ECU
3  |  foreach runnable in the sofware component do
4  |  |  randomly assign it to an Core on the ECU
5  |  end
6  end
7  Compute current cost;
8  while current temperature > minimum temperature do
9  |  for step := 1..max steps per temperature do
10 |  |  Randomly choose a strategy;
11 |  |  Generate new solution from the current solution;
12 |  |  Compute new cost;
13 |  |  if new cost < curent cost then
14 |  |  |  current solution = new solution
15 |  |  else
16 |  |  |  Choose a random number r ∈ [0, 1);
17 |  |  |  if e^((old cost−new cost)/current temperature) > r then
18 |  |  |  |  current solution = new solution;
19 |  |  |  else
20 |  |  |  end
21 |  |  end
22 |  end
23 |  current temperature = current temperature ∗ cooling factor
24 end
```

Algorithm 3.2: Simulated Annealing for mapping problem

3.6.1 Implementation details

The implementation of the tool has been done in C# programming language[1] and runs on top of Microsoft's .NET Framework[2]. Since C# is an object-oriented programming language, it is based on the concept of classes that groups together data known as attributes and code known as methods that are relying on the attributes to implement their functionality. Classes that are semantically related can be grouped together into software packages. From implementation point of view, the mapping tool can be split into four major software packages (fig. 3.6). The *HardwareModel* package contains all the classes responsible for

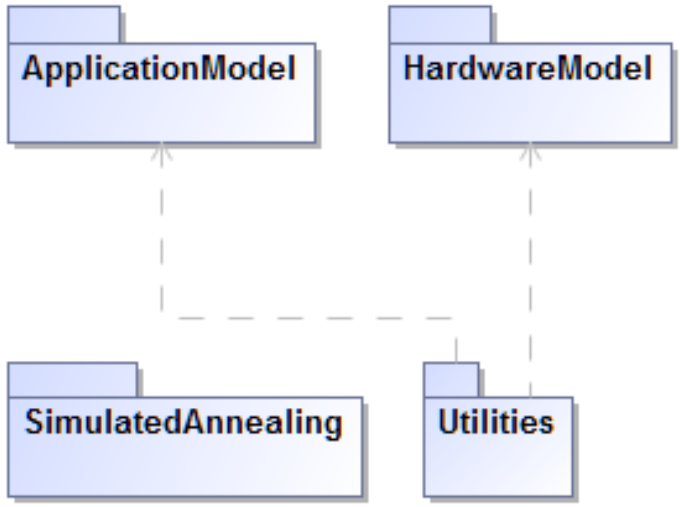


Figure 3.6: Main software packages

handling the hardware model. Similarly, the *ApplicationModel* package groups the classes for handling the application model. *SimulatedAnnealing* package contains the classes and data structures needed for the simulated algorithm 3.2. The *Utilities* package contains support classes such as the ones that are reading the application and hardware model files.

[1] `https://msdn.microsoft.com/en-us/library/618ayhy6(v=vs.71).aspx`
[2] `https://www.microsoft.com/net`

The main classes that are associated with the application model are presented in fig. 3.11. There is an equivalent class for each of the concepts introduced in section 2.1: *software component, runnable, Os-Task, Os-Application.* An instance of one of this classes is equivalent to a "real" *software component, runnable, etc.* When each instance of the *software component* and *runnable* is created from the application model, an unique ID is associated with it. The ID is just an integer that is used by all the generation strategies for constructing the mapping solution.

The main classes that are associated with the hardware model are presented in fig. 3.12. As in the case of the application model, there is an equivalent class for each of the concepts that where introduced in section 2.2: ECU, core, inter-core and inter-ECU communication. Again, each instance of ECU and core classes has an unique ID that is being used by the generation strategies to construct a mapping solution.

The fig. 3.10 shows the main classes that were created for the implementation of the simulated annealing algorithm. The strategy design pattern [GHJV95] has been used for defining the transformation classes. We have one class for each generation strategy introduced in 3.6: **MoveSoftwareComponentStrategy S.1**, **MoveRunnableBetweenCoresStrategy S.2** and **MoveRunnableIntoTaskStrategy S.3**. In the case when we have only one ECU with multiple-cores moving *software components* does not make sense and only the transformations **S.2** and **S.3** are used.

At each iteration of the simulated annealing algorithm one of the generation strategy is chosen with a given probability. We have decided that the probability of choosing a mapping of *runnables* to cores or mapping of *runnables* to *Os-Tasks* should be twice as high compared to the mapping of *software components* to the ECUs. The motivation is that we do not want to move too often the *software components* since the move will also imply that all the *runnables* inside them will have to be remapped to the new cores of the ECU.

The **SolutionModel** class contains the mapping of *software components* to ECUs, *runnables* to cores, *runnables* to *Os-Tasks* and *Os-Tasks* to *Os-Applications.* Except for the *Os-Task* to *Os-Application* grouping, each of the above mentioned mappings is defined using a data structure named dictionary (see [Ski08]). Each member of the dictionary has an unique key id (used for referencing) and an associated value.

For our problem we have defined the following dictionaries:

- **softwareComponentToECUMap**. Maps a *software component instance ID* (plays the role of the key) to a *ECU instance ID* (plays the role of the value).
- **runnableToCoreMap**. Maps *a runnable instance ID* (plays the role of the key) to a *core instance ID* (plays the role of the value).
- **runnableToTaskMapping**. Groups together two *runnable* instance IDs into the same *Os-Task*. In this case, the key represents the *runnable* that needs to be grouped and the value represents the *runnable* with which it will be grouped. We consider that each *runnable instance* is mapped into its own *Os-Task*, therefore the value of each key in the dictionary can be treated as a task ID.

The motivation of choosing this particular data structure is that the implementation of it is based on a hash table, data structure that allows accessing a key close to O(1) time [3]. This is useful in our case since each generation strategy will access/modify each dictionary many times at each temperature level.
In the following, a couple of examples will be given to show how **runnableToTaskMapping** dictionary is used for grouping the *runnables* together. First, if we have two *runnable* instances, one with ID = 1 and the other with the ID 2 and we group them together into an *Os-Task* with the ID equal to the *runnable* with ID = 2, then we will have that **runnableToTaskMapping[2]=1**. This can be seen as a graph where the task ID is the parent node and all the *runnables* grouped inside are the children having an arrow pointing to it fig. 3.7.

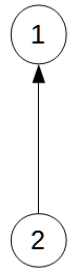


Figure 3.7: Grouping of two runnables

[3] https://msdn.microsoft.com/en-us/library/xfhwa508%28v=vs.110%29.aspx

Let's consider a more complex case where we have 4 *runnables* instances with the IDs: 1,2,3,4. They are all grouped into the same *Os-Task* and its ID is the same as for the *runnable* instance with ID 1 (fig. 3.8). The dictionary **runnableToTaskMapping** will have the following configuration:

runnableToTaskMapping[1]=1,runnableToTaskMapping[2]=1,
runnableToTaskMapping[3]=1,runnableToTaskMapping[4]=1.

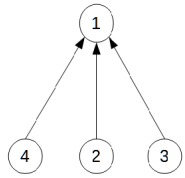


Figure 3.8: Grouping of four runnables

Now, assume that a *runnable* instance with the ID = 1 is mapped to another core. We have decided that in this case, we should not "break" the group, but use one of the remaining *runnable* IDs as the new *Os-Task* ID. Therefore, the *runnable* instances with ID 2,3,4 are grouped together and the *Os-Task* ID is equal to the *runnable* with the ID 4. The *runnable* with ID 1 is grouped in its own *Os-Task* (fig. 3.9). After this transformation, the dictionary **runnableToTaskMapping** will have the following configuration:

runnableToTaskMapping[1]=1,runnableToTaskMapping[2]=4,
runnableToTaskMapping[3]=4,runnableToTaskMapping[4]=4.

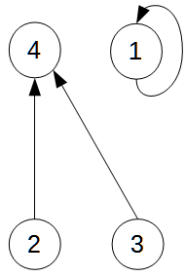


Figure 3.9: Grouping of four runnables after one runnable is remapped

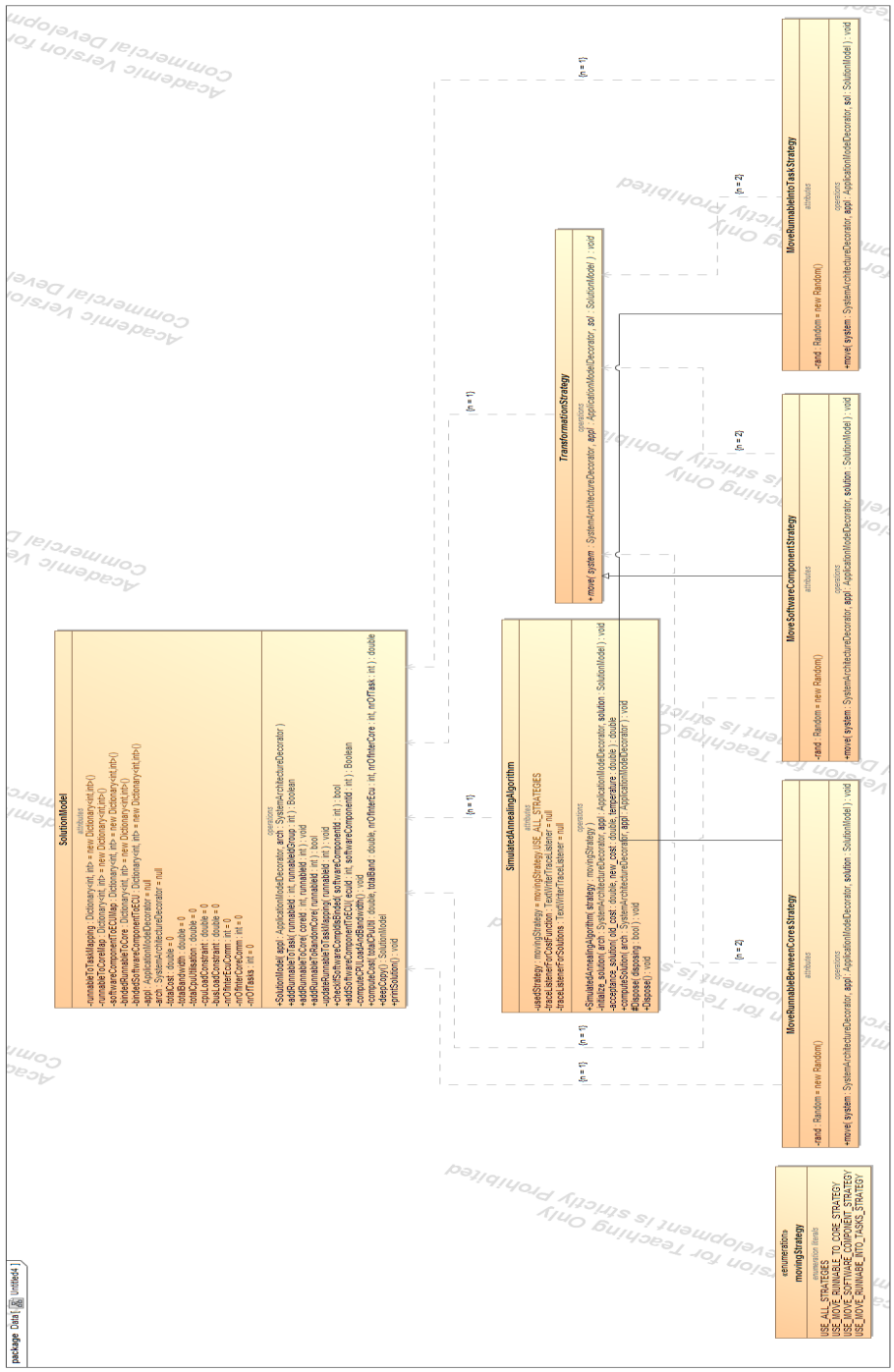


Figure 3.10: Classes from Simulated Annealing package

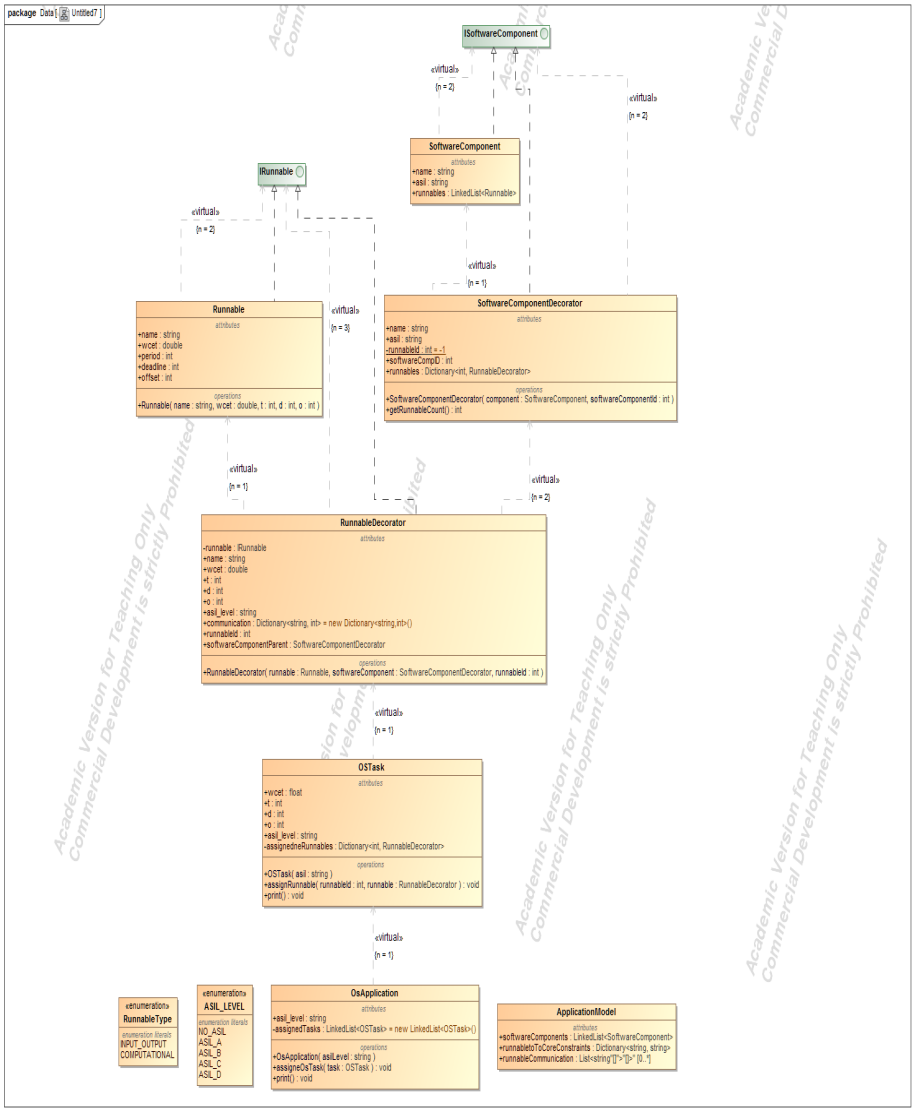


Figure 3.11: Classes from Application Model package

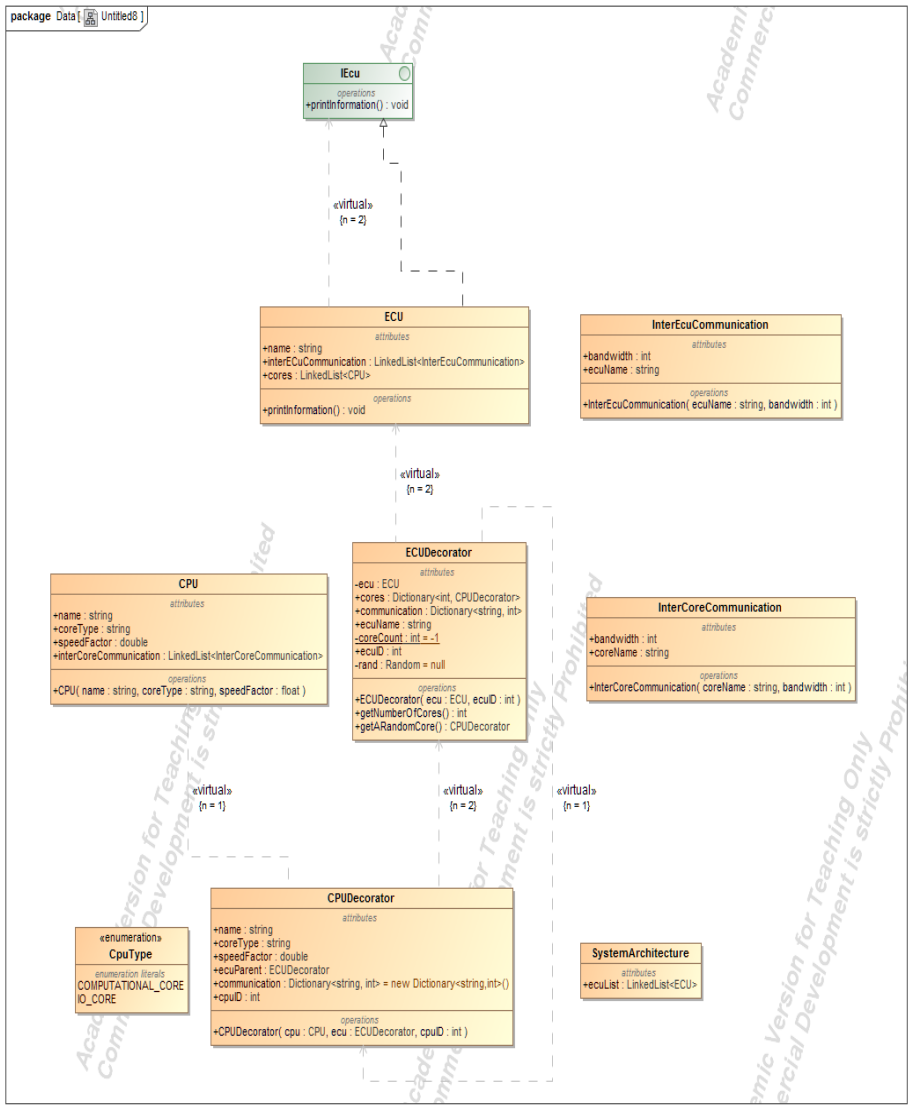


Figure 3.12: Classes from Hardware model

3.7 Test cases

Three application models were used to test the mapping tool. The first application was inspired from [ATPK+11]. The second one was adapted from an automotive use case presented in [CDKM02]. The last use case was provided by Volvo Advanced Technology & Research in Götheborg, Sweden.

3.7.1 Map tool debugging and testing

The unit test framework provided by *Microsoft's Visual Studio 2012 IDE* [4] has been used to test and validate the implementation. The tests were designed in such a way that they will automatically run after each source code change (revision) of the project. Using this strategy, we could spot new bugs earlier and fix them. The unit tests were developed mainly to check the generation strategies applied for the solution (section 3.6.1). In combination with the unit tests, assert statements were introduced in the code to test different conditions like the core utilization should be a valid double number, the *runnables* that are mapped into the same *Os-Task* have to be on the same core (see listing 3.1), etc.

Listing 3.1: Example of assert statement

```
1 System.Diagnostics.Debug.Assert(runnableToCoreMap[value.Key] ==
      runnableToCoreMap[group.Key], "The runnables are not on the
      same core");
```

An example of unit test execution is shown in fig. 3.13.

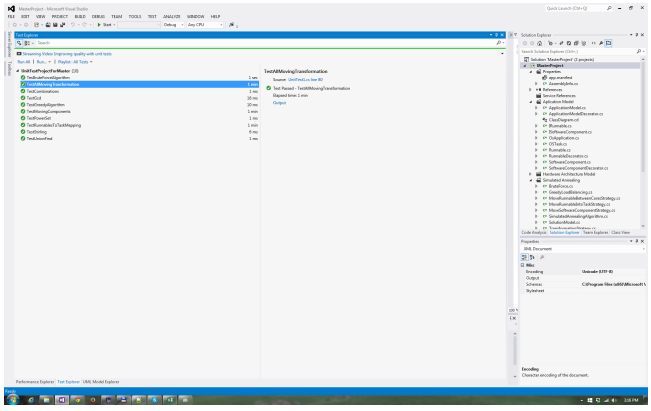


Figure 3.13: Unit Tests execution

[4] https://msdn.microsoft.com/en-us/library/hh694602.aspx

3.7.2 Test application and architecture model

The application model in fig. 3.14 represents a vehicle cruise control system and it is composed of two *software components*. The first *software component* named

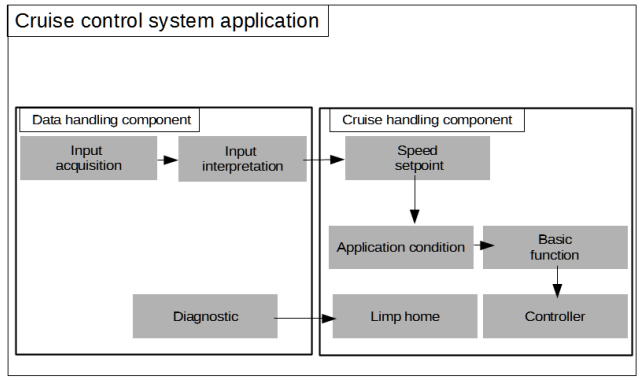


Figure 3.14: Cruise control application

Data handling has three *runnables*:

- *Input acquisition* - responsible for acquiring sensor data.
- *Input interpretation* - interprets the sensor data to determine the driver's desire wish.
- *Diagnostics* - responsible for detecting errors and inconsistency in the sensor data.

The second *software component* named *Cruise handling controller* contains six *runnables*:

- *Speed set-point* - responsible for computing the speed desired by the driver.
- *Application condition, Basic function* - responsible for computing cruise control states and transitions.
- *Limp home* - decides which action to take when an error has been detected.
- *Controller* - proportional-integral controller that maintains the vehicle speed.

The table 3.2 contains the period, WCET and ASIL level of all the *runnables* in the *software components*. In the table 3.3 it is shown the number of bytes that are being send between the *runnables*.

Table 3.2: Runnable information

Runnable	WCET(ms)	Period(ms)	Deadline(ms)	ASIL
Input aquisition	0.5	10	10	A
Input interpretation	1	10	10	A
Diagnostic	1.5	10	10	A
Speed Setpoint	1	10	10	No ASIL
Limp home	1.5	10	10	No ASIL
Basic function	2.5	10	10	No ASIL
Controller	3	10	10	No ASIL

Table 3.3: Runnable signal information

Sender runnable	Receiver runnable	Data bytes
Input aquisition	Input interpretation	2
Input interpretation	Speed setpoint	4
Diagnostic	Limp home	8
Application condition	Basic function	4
Speed setpoint	Application condition	2
Basic function	Controller	8

The architecture model consists of two ECUs. One ECU has a single core and the second-one is composed of three cores (fig. 3.15). The inter-ECU and inter-core bandwidth are presented in table 3.5 and table 3.4.

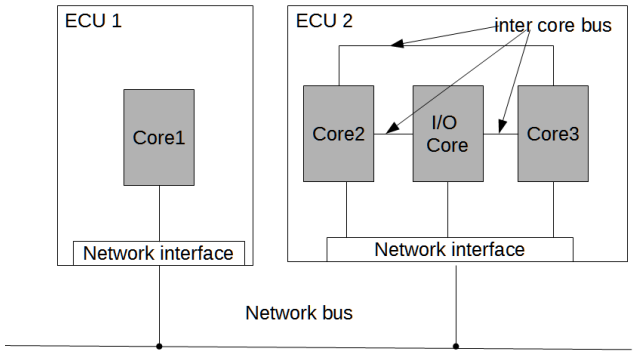


Figure 3.15: Hardware architecture model

Table 3.4: Inter core bandwidth

Core Name	Core name	Bandwidth (bytes/second)
Core1	Core2	10000
Core1	I/O core	10000
Core2	I/O core	10000

Table 3.5: Inter ECU bandwidth

ECU name	ECU name	Bandwidth(bytes/second)
ECU1	ECU2	50000

3.7.3 Automotive application and architecture model

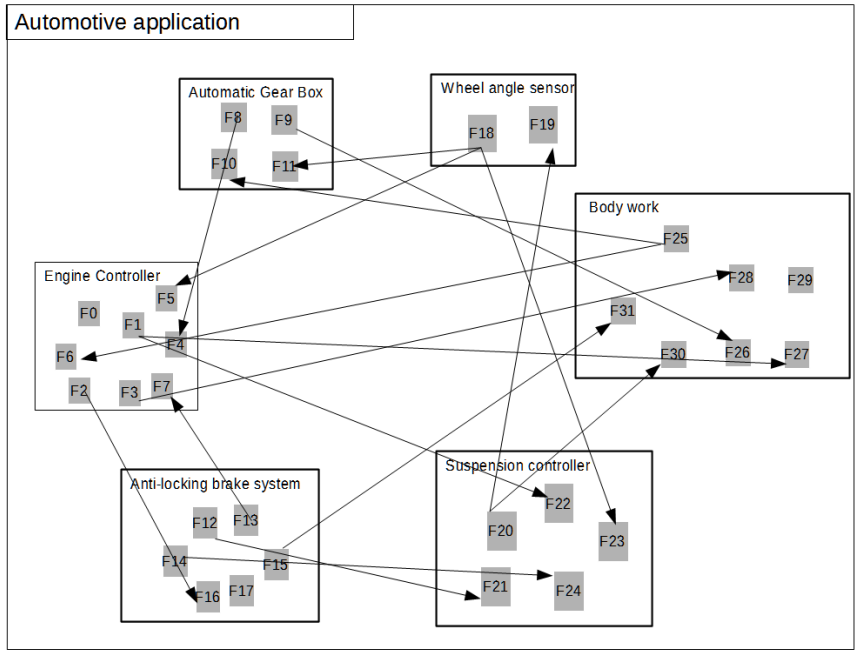


Figure 3.16: Automotive application

The application model has been inspired from [CDKM02]. In this case we have six *software components* with a total of thirty one *runnables* with arrows between them representing the data signals that are send (fig. 3.16).

The *runnable's* configuration is presented in table 3.6. The table 3.8 shows the *runnables* that are exchanging data signals together with the amount of bytes that are being send.

- *Engine Controller.* Composed of seven *runnables*: $F0 - F7$
- *Automatic Gear Box.* Composed of four *runnables*: $F8 - F11$
- *Anti-locking brake.* Composed of six *runnables*: $F12 - F17$
- *Wheel angle sensor.* Composed of two *runnables*: $F18 - F19$
- *Suspension controller.* Composed of five *runnables*: $F20 - F24$
- *Body work.* Composed of seven *runnables*: $F25 - F31$

Table 3.6: Runnable information for automotive application

Runnable	WCET(ms)	Period(ms)	Deadline(ms)	ASIL
F1	2	10	10	C
F2	2	20	20	C
F3	2	100	100	C
F4	2	15	15	C
F5	2	14	14	C
F6	2	50	50	C
F7	2	40	40	C
F8	2	15	15	D
F9	2	15	15	D
F10	2	50	50	D
F11	2	14	14	D
F12	1	20	20	D
F13	2	20	20	D
F14	1	15	15	D
F15	2	100	100	D
F16	1	20	20	D
F17	2	14	14	D
F18	4	14	14	B
F19	4	20	20	B
F20	1	20	20	C
F21	1	20	20	C
F22	1	10	10	C
F23	2	14	14	C
F24	2	15	15	C
F25	2	50	50	A

F26	2	50	50	A
F27	2	10	10	A
F28	2	100	100	A
F29	2	40	40	A
F30	2	20	20	A
F31	2	100	100	A

Table 3.8: Runnable signal information for automotive application

Runnable sender	Runnable receiver	Data bytes
F8	F4	2
F13	F7	5
F2	F16	3
F18	F5	3
F18	F11	3
F18	F23	3
F20	F19	4
F12	F21	5
F1	F22	8
F14	F24	4
F25	F10	5
F25	F6	5
F9	F26	5
F1	F27	8
F3	F28	7
F20	F30	4
F15	F31	1

The architecture model chosen consists of two ECUs with three cores each (fig. 3.17). The values for the inter-core and inter-ECU communication bandwidth are presented in table 3.9 and table 3.10.

Table 3.9: Inter core bandwidth

Core Name	Core name	Bandwidth (bytes/second)
Core1	Core2	100000
Core1	I/O core 1	100000
Core2	I/O core 1	100000
Core3	Core4	100000
Core3	I/O core 2	100000
Core4	I/O core 2	100000

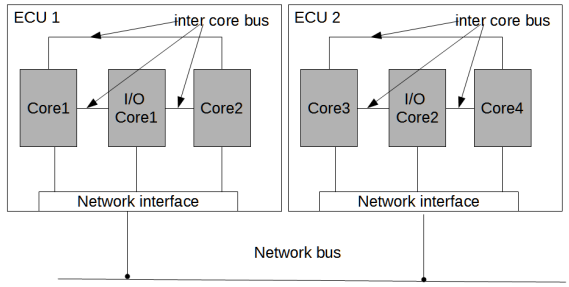


Figure 3.17: Automotive hardware architecture

Table 3.10: Inter ECU bandwidth

ECU name	ECU name	Bandwidth(bytes/second)
ECU1	ECU2	500000

3.7.4 Volvo application and architecture model

The use case from Volvo consists of 50 *software components* with 75 *runnables* in total. It has been provided by Power-train division where the development is focused on controlling the automobile's engine transmission , suspension, wheels, etc. Due to high number of *software components* and *runnables*, the application model is described in JSON[5] format in appendix A.1. The hardware model consists of one ECU with 3 cores as requested by Volvo (fig. 3.18). The inter-core communication bandwidth are presented in table 3.11.

Table 3.11: Inter core bandwidth

Core Name	Core name	Bandwidth (bytes/second)
Core1	Core2	500000
Core1	I/O core	500000
Core2	I/O core	500000

[5] `http://www.w3schools.com/json/`

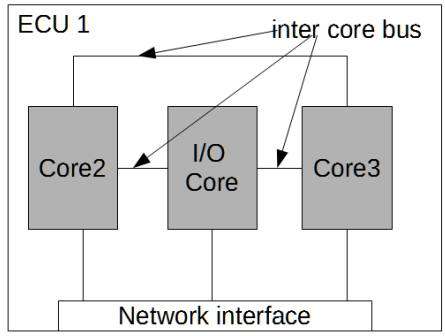


Figure 3.18: Architecture model for Volvo use case

3.8 Experimental results

The test application and the architecture model presented in section 3.7.2 has been used for the entire development process for debugging and proof of concept. The outcome of the simulated annealing heuristic depends on the following parameters: initial and final temperature, the number of iterations per temperature and the cooling factor. For the parameters in table 3.12 we have chosen the values as recommended in [Ski08].

Table 3.12: Simulated Annealing Parameters

Parameter	Value
Initial Temperature	1
Final Temperature	0.00001
Cooling Factor	0.95

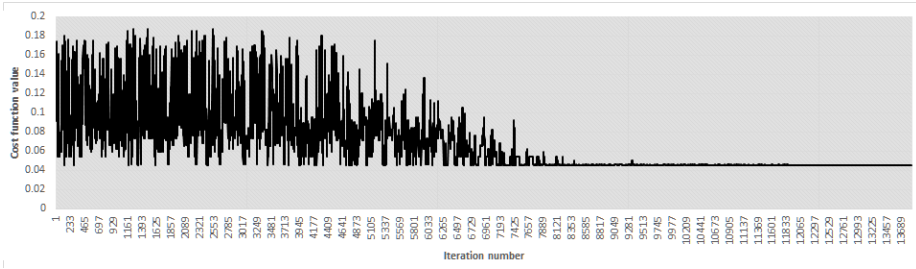


Figure 3.19: Cost function values

Given the test application as input for the tool, the chart in fig. 3.19 plots the cost function values against each iteration per temperature.

At the beginning, when the temperature is high, the cost values varies widely as the algorithm allows more "jumps". As the temperature is cooling down, the cost value jumps decrease when the algorithm is reaching a solution with the minimum value.
The cost function values in fig. 3.19 where obtained having the number of steps per temperature set to **100**. According to [Ski08], the typical values for the number of iterations accepted before the temperature is cooled down is in the range between **100** and **1000**. As for the $WCET_{communication}$ overhead values (see section 2.3.3), we have used the values in the table 3.13. Also, for the cost function 3.6, we have set the weights W_1 and W_2 to **0.5**, the maximum core utilization to **0.69** and the penalty factor equal to **1000**.

Table 3.13: $WCET_{communication}$ overhead for test application

.

Overhead	Value(ms)
α	0.01
β_0	0.02
β_1	0.03
γ	0.04
θ	0.06

3.8.1 Test application

Given the test application and the hardware architecture from section 3.7.2, a possible solution with associated cost of **0.045** is presented in fig. 3.20. Both *software components* are mapped into different ECUs with all constraints regarding core, inter-core and inter-ECU bandwith utilization met. Since our application test is quite small in the number of *runnables* and *software components*, we have used the backtracking solution (see section 3.4) to find the optimal partitioning. The algorithm outputs that the optimal solution has the cost **0.045**, the same as the one obtained by the simulated annealing approach.

ECU2 ID: 0
SofwareComponent1 ID :0
ECU1 ID: 1
SofwareComponent2 ID :1
Core3 ID: 0
Input aquisition ID: 0
Input interpretation ID: 1
Diagnostic ID: 2
Core2 ID: 2
Speed setpoint ID: 3
Application condition ID: 5
Core1 ID: 1
Limp home ID: 4
I/O Core ID: 3
Basic function ID: 6
Controller ID: 7
Number of tasks :7
Group = 0:

```
0
Group = 1:
1
Group = 2:
2
Group = 3:
3
Group = 4:
4
Group = 5:
5
Group = 6:
6 7
Core3 ID: 0
Os application : ASIL_A
Os task: ASIL_A
runnable id: 0 runnable name : Input aquisition runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 1 runnable name : Input interpretation runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 2 runnable name : Diagnostic runnable asil level : ASIL_A
Core1 ID: 1
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 4 runnable name : Limp home runnable asil level : NO_ASIL
Core2 ID: 2
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 3 runnable name : Speed setpoint runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 5 runnable name : Application condition runnable asil level : NO_ASIL
I/O Core ID: 3
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 6 runnable name : Basic function runnable asil level : NO_ASIL
runnable id: 7 runnable name : Controller runnable asil level : NO_ASIL
```

Figure 3.20: Solution for the test application and architecture model

Furthermore, we wanted to test the impact of parameters W_1 and W_2 on solutions obtained. Three cases are taken into consideration: ($W_1 = 1$ and $W_2 = 0$), ($W_1 = 0$ and $W_2 = 1$) and ($W_1 = 0.5$ and $W_2 = 0.5$).

In the case when we have $W_1 = \mathbf{1}$ and we ignore the bandwidth utilization minimization, one possible solution with an associated cost of **0.001072** is presented in fig. 3.21. All the constraints regarding core, inter-ECU/inter-core bandwidth utilization are met. The first observation to note is an increase in inter-core communication. Compared with the solution in fig. 3.20 we have three pairs of runnables that communicate with each other instead of one. One explanation is that the simulated annealing has searched for a solution where all *runnables* were distributed among all cores such that the utilization per each core is closer to the mean value.

```
inter ecu communication is :2
inter core communication is :3
ECU2 ID: 0
SofwareComponent1 ID :0
ECU1 ID: 1
SofwareComponent2 ID :1
Core3 ID: 0
Input aquisition ID: 0
Input interpretation ID: 1
Diagnostic ID: 2
Core1 ID: 1
Speed setpoint ID: 3
```

```
Basic function ID: 6
Core2 ID: 2
Limp home ID: 4
Application condition ID: 5
I/O Core ID: 3
Controller ID: 7
Number of tasks :7
Group = 0:
0
Group = 1:
1
Group = 2:
2
Group = 3:
3
Group = 4:
4 5
Group = 6:
6
Group = 7:
7
Core3 ID: 0
Os application : ASIL_A
Os task: ASIL_A
runnable id: 0 runnable name : Input aquisition runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 1 runnable name : Input interpretation runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 2 runnable name : Diagnostic runnable asil level : ASIL_A
Core1 ID: 1
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 3 runnable name : Speed setpoint runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 6 runnable name : Basic function runnable asil level : NO_ASIL
Core2 ID: 2
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 4 runnable name : Limp home runnable asil level : NO_ASIL
runnable id: 5 runnable name : Application condition runnable asil level : NO_ASIL
I/O Core ID: 3
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 7 runnable name : Controller runnable asil level : NO_ASIL
```

Figure 3.21: Solution for the test application and architecture model in the case of ignoring utilization bandwidth

In the case when we have $W_2 = \mathbf{1}$ and we ignore the variance of core utilization, one possible solution with an associated cost of **0.06** is presented in fig. 3.22. Compared to the previous ones, now the simulated annealing approach has mapped all the *software components* into one ECU such that the inter-ECU bandwidth utilization is 0. Furthermore, the *runnables* on the *cores* were mapped in such way that only one pair of *runnables* is sending data over inter-core network. Again, the solution meets all the constraints regarding maximum core and bandwidth utilization.

```
total cpu is :0.060156
total bandwidth is :0.04
inter ecu communication is :0
inter core communication is :1
ECU1 ID: 1
SofwareComponent1 ID :0
SofwareComponent2 ID :1
I/O Core ID: 3
Input aquisition ID: 0
Input interpretation ID: 1
Speed setpoint ID: 3
Application condition ID: 5
```

```
Core1 ID: 1
Diagnostic ID: 2
Limp home ID: 4
Core2 ID: 2
Basic function ID: 6
Controller ID: 7
Number of tasks :8
Group = 0:
0
Group = 1:
1
Group = 2:
2
Group = 3:
3
Group = 4:
4
Group = 5:
5
Group = 6:
6
Group = 7:
7
Core3 ID: 0
Core1 ID: 1
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 4 runnable name : Limp home runnable asil level : NO_ASIL
Os application : ASIL_A
Os task: ASIL_A
runnable id: 2 runnable name : Diagnostic runnable asil level : ASIL_A
Core2 ID: 2
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 6 runnable name : Basic function runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 7 runnable name : Controller runnable asil level : NO_ASIL
I/O Core ID: 3
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 3 runnable name : Speed setpoint runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 5 runnable name : Application condition runnable asil level : NO_ASIL
Os application : ASIL_A
Os task: ASIL_A
runnable id: 0 runnable name : Input aquisition runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 1 runnable name : Input interpretation runnable asil level : ASIL_A
```

Figure 3.22: Solution for the test application and architecture model in the case of ignoring core utilization

3.8.2 Automotive application

We constructed the application model in fig. 3.16 that consists of six *software components* of different ASIL levels with a total number of 31 *runnables*. For the architecture model, we have chosen two ECUs with 3 cores each fig. 3.17 . Due to high number of *runnables*, the straight forward backtracking algorithm fails to find the optimal cost reporting an "out of memory exception" while trying to construct a solution. For this application model, we have increased to **500** the number of iterations before the temperature is lowered down. For the weights of the cost function three cases where chosen: one where we have equal weights and two cases when one weight is ignored (has value 0). A number of **5** tests have been performed for each case.

We were interested to see how the parameter weights affect the solutions obtained in terms of the number of *runnable* pairs that communicate inter-core and inter-ECU and number of *Os-Tasks* obtained.

Table 3.14: Cost values when the overall communication bandwidth is ignored

Cost	Runnable group count communicating inter-ECU	Runnable group count communicating inter-Core	Os-Tasks Count
0.00119	8	8	26
0.00101	11	5	23
0.00119	11	5	24
0.00067	11	4	22
0.00094	11	6	26

When the overall communication bandwidth is ignored ($W_2 = 0$), the simulated annealing will search for a solution where the variance of the core utilization is small. Given the results in table 3.14 it can be seen that from a total of 17 pairs of runnables that are exchanging signals (table 3.8), almost all of them are mapped such that they are sending data using either inter-core or inter-ECU communication.

A solution associated with the cost of **0.00067** is presented in fig. 3.23. We can observe that the simulated annealing approach has evenly mapped the *software components* on the ECU1 and ECU2 and the number of *runnables* per core is no more than 4-6. The constraints regarding the core and bus bandwidth utilization are also met. This is normal since we ignore the bandwidth utilization, and solutions with high number of *runnables* exchanging data between cores and ECUs are accepted as long as the maximum bandwidth utilization are not exceeded.

```
inter ecu communication is :11
inter core communication is :4
ECU1 ID: 1
Engine Controller ID :0
Automatic Gear Box ID :1
Anti-locking brake ID :2
ECU2 ID: 0
Wheel angle sensor ID :3
Suspension controller ID :4
Body work ID :5
Core1 ID: 3
F1 ID: 0
F2 ID: 1
F6 ID: 5
F10 ID: 9
F17 ID: 16
I/O Core1 ID: 5
F3 ID: 2
F5 ID: 4
F8 ID: 7
F11 ID: 10
F12 ID: 11
F15 ID: 14
Core2 ID: 4
F4 ID: 3
```

F7 ID: 6
F9 ID: 8
F13 ID: 12
F14 ID: 13
F16 ID: 15
Core4 ID: 1
F18 ID: 17
F20 ID: 19
F25 ID: 24
F30 ID: 29
Core3 ID: 0
F19 ID: 18
F22 ID: 21
F24 ID: 23
F28 ID: 27
F31 ID: 30
I/O Core2 ID: 2
F21 ID: 20
F23 ID: 22
F26 ID: 25
F27 ID: 26
F29 ID: 28
Number of tasks :22
Group = 0:
0
Group = 1:
1 5
Group = 4:
2 4
Group = 3:
3
Group = 6:
6
Group = 7:
7
Group = 15:
8 12 15
Group = 9:
9
Group = 10:
10 11
Group = 13:
13
Group = 14:
14
Group = 16:
16
Group = 17:
17
Group = 18:
18
Group = 19:
19
Group = 22:
20 22
Group = 21:
21
Group = 23:
23
Group = 24:
24 29
Group = 25:
25 26 28
Group = 27:
27
Group = 30:
30
Core3 ID: 0
Os application : ASIL_A
Os task: ASIL_A
runnable id: 27 runnable name : F28 runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 30 runnable name : F31 runnable asil level : ASIL_A
Os application : ASIL_B
Os task: ASIL_B
runnable id: 18 runnable name : F19 runnable asil level : ASIL_B
Os application : ASIL_C
Os task: ASIL_C
runnable id: 21 runnable name : F22 runnable asil level : ASIL_C
Os task: ASIL_C
runnable id: 23 runnable name : F24 runnable asil level : ASIL_C
Core4 ID: 1
Os application : ASIL_A
Os task: ASIL_A

```
runnable id: 24 runnable name : F25 runnable asil level : ASIL_A
runnable id: 29 runnable name : F30 runnable asil level : ASIL_A
Os application : ASIL_B
Os task: ASIL_B
runnable id: 17 runnable name : F18 runnable asil level : ASIL_B
Os application : ASIL_C
Os task: ASIL_C
runnable id: 19 runnable name : F20 runnable asil level : ASIL_C
I/O Core2 ID: 2
Os application : ASIL_A
Os task: ASIL_A
runnable id: 25 runnable name : F26 runnable asil level : ASIL_A
runnable id: 26 runnable name : F27 runnable asil level : ASIL_A
runnable id: 28 runnable name : F29 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 20 runnable name : F21 runnable asil level : ASIL_C
runnable id: 22 runnable name : F23 runnable asil level : ASIL_C
Core1 ID: 3
Os application : ASIL_C
Os task: ASIL_C
runnable id: 0 runnable name : F1 runnable asil level : ASIL_C
Os task: ASIL_C
runnable id: 1 runnable name : F2 runnable asil level : ASIL_C
runnable id: 5 runnable name : F6 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 9 runnable name : F10 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 16 runnable name : F17 runnable asil level : ASIL_D
Core2 ID: 4
Os application : ASIL_C
Os task: ASIL_C
runnable id: 3 runnable name : F4 runnable asil level : ASIL_C
Os task: ASIL_C
runnable id: 6 runnable name : F7 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 8 runnable name : F9 runnable asil level : ASIL_D
runnable id: 12 runnable name : F13 runnable asil level : ASIL_D
runnable id: 15 runnable name : F16 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 13 runnable name : F14 runnable asil level : ASIL_D
I/O Core1 ID: 5
Os application : ASIL_C
Os task: ASIL_C
runnable id: 2 runnable name : F3 runnable asil level : ASIL_C
runnable id: 4 runnable name : F5 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 7 runnable name : F8 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 10 runnable name : F11 runnable asil level : ASIL_D
runnable id: 11 runnable name : F12 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 14 runnable name : F15 runnable asil level : ASIL_D
```

Figure 3.23: Solution for the automotive application and architecture model in the case of ignoring bandwidth utilization

If we ignore the core utilization variance ($W_1 = 0$), the results in table 3.15 show that in all of the five cases, the simulated annealing finds solutions where the *runnables* communicate only using inter-ECU network. Because of the high number of *runnables* exchanging data signals (see fig. 3.16), they could not be mapped onto one ECU due to the high core utilization resulted, therefore some of them ended being mapped to the cores of the other ECU, resulting in an increase in the inter-ECU communication. A solution with the associated cost of **0.0042** is presented in fig. 3.24 where even though it meets all the bandwidth and core utilization constraints, the number of *runnables* assigned to the I/O Core1 is almost double compared to the other cores. This result is normal in this case since the core utilization variance does not have a impact in the cost.

Table 3.15: Cost values when the core utilization variance is ignored

Cost	Runnable group count communicating inter-ECU	Runnable group count communicating inter-Core	Os-Tasks Count
0.00388	9	0	24
0.00424	11	0	16
0.00388	9	0	21
0.00388	9	0	22
0.00424	11	0	17

Cost : 0.00424380952380952
inter ecu communication is :11
inter core communication is :0
ECU1 ID: 1
Engine Controller ID :0
Suspension controller ID :4
Body work ID :5
ECU2 ID: 0
Automatic Gear Box ID :1
Anti-locking brake ID :2
Wheel angle sensor ID :3
Core2 ID: 4
F1 ID: 0
F4 ID: 3
F22 ID: 21
F27 ID: 26
Core1 ID: 3
F2 ID: 1
F5 ID: 4
F23 ID: 22
F26 ID: 25
F31 ID: 30
I/O Core1 ID: 5
F3 ID: 2
F6 ID: 5
F7 ID: 6
F20 ID: 19
F21 ID: 20
F24 ID: 23
F25 ID: 24
F28 ID: 27
F29 ID: 28
F30 ID: 29
Core3 ID: 0
F8 ID: 7
F10 ID: 9
F12 ID: 11
F17 ID: 16
Core4 ID: 1
F9 ID: 8
F13 ID: 12
F14 ID: 13
F15 ID: 14
F19 ID: 18
I/O Core2 ID: 2
F11 ID: 10
F16 ID: 15
F18 ID: 17
Number of tasks :16
Group = 3:
0 3 21
Group = 4:
1 4 22
Group = 2: 2 **Group = 20:** 5 6 19 20 23 **Group = 16:**
7 16 **Group = 8:**
8
Group = 11:
9 11
Group = 10:
10 15
Group = 12:

```
12 14
Group = 13:
13
Group = 17:
17
Group = 18:
18
Group = 24:
24
Group = 25:
25 30
Group = 26:
26
Group = 28:
27 28 29
Core3 ID: 0
Os application : ASIL_D
Os task: ASIL_D
runnable id: 7 runnable name : F8 runnable asil level : ASIL_D
runnable id: 16 runnable name : F17 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 9 runnable name : F10 runnable asil level : ASIL_D
runnable id: 11 runnable name : F12 runnable asil level : ASIL_D
Core4 ID: 1
Os application : ASIL_B
Os task: ASIL_B
runnable id: 18 runnable name : F19 runnable asil level : ASIL_B
Os application : ASIL_D
Os task: ASIL_D
runnable id: 8 runnable name : F9 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 12 runnable name : F13 runnable asil level : ASIL_D
runnable id: 14 runnable name : F15 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 13 runnable name : F14 runnable asil level : ASIL_D
I/O Core2 ID: 2
Os application : ASIL_B
Os task: ASIL_B
runnable id: 17 runnable name : F18 runnable asil level : ASIL_B
Os application : ASIL_D
Os task: ASIL_D
runnable id: 10 runnable name : F11 runnable asil level : ASIL_D
runnable id: 15 runnable name : F16 runnable asil level : ASIL_D
Core1 ID: 3
Os application : ASIL_A
Os task: ASIL_A
runnable id: 25 runnable name : F26 runnable asil level : ASIL_A
runnable id: 30 runnable name : F31 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 1 runnable name : F2 runnable asil level : ASIL_C
runnable id: 4 runnable name : F5 runnable asil level : ASIL_C
runnable id: 22 runnable name : F23 runnable asil level : ASIL_C
Core2 ID: 4
Os application : ASIL_A
Os task: ASIL_A
runnable id: 26 runnable name : F27 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 0 runnable name : F1 runnable asil level : ASIL_C
runnable id: 3 runnable name : F4 runnable asil level : ASIL_C
runnable id: 21 runnable name : F22 runnable asil level : ASIL_C
I/O Core1 ID: 5
Os application : ASIL_A
Os task: ASIL_A
runnable id: 24 runnable name : F25 runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 27 runnable name : F28 runnable asil level : ASIL_A
runnable id: 28 runnable name : F29 runnable asil level : ASIL_A
runnable id: 29 runnable name : F30 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 2 runnable name : F3 runnable asil level : ASIL_C
Os task: ASIL_C
runnable id: 5 runnable name : F6 runnable asil level : ASIL_C
runnable id: 6 runnable name : F7 runnable asil level : ASIL_C
runnable id: 19 runnable name : F20 runnable asil level : ASIL_C
runnable id: 20 runnable name : F21 runnable asil level : ASIL_C
runnable id: 23 runnable name : F24 runnable asil level : ASIL_C
```

Figure 3.24: Solution for the automotive application and architecture model in the case of ignoring core utilization variance

When we have equal weights for core utilization variance and bus bandwidth, there is a slight increase in the number of *runnables* that communicate over ECU (table 3.16) compared with the results obtained in table 3.15. As in the case when the core utilization variance was ignored, not all the *runnables* could be mapped onto one ECU without having the core utilization constraint broken, therefore part of *runnables* were mapped on the other ECU increasing the inter-ECU communication. It appears that for this application model, the core utilization constraint has an impact on the communication bandwidth. A solution with an associated cost of **0.00386** is presented in fig. 3.25 where the number of *runnables* per core is between 4-7 which is almost as good as the solution obtained when the communication bandwidth were ignored (fig. 3.23).

Table 3.16: Cost values when the weights for overall bandwidth utilization and core utilization variance are the same

Cost	Runnable group count communicating inter-ECU	Runnable group count communicating inter-Core	Os-Tasks Count
0.00427	14	0	14
0.00337	11	0	17
0.00428	14	0	17
0.00423	10	1	23
0.00386	11	0	19

inter ecu communication is :11
inter core communication is :0
ECU2 ID: 0
Engine Controller ID :0
Anti-locking brake ID :2
Wheel angle sensor ID :3
ECU1 ID: 1
Automatic Gear Box ID :1
Suspension controller ID :4
Body work ID :5
Core3 ID: 0
F1 ID: 0
F2 ID: 1
F16 ID: 15
F19 ID: 18
Core4 ID: 1
F3 ID: 2
F4 ID: 3
F6 ID: 5
F7 ID: 6
F12 ID: 11
F13 ID: 12
F17 ID: 16
I/O Core2 ID: 2
F5 ID: 4
F14 ID: 13
F15 ID: 14
F18 ID: 17
I/O Core1 ID: 5
F8 ID: 7
F9 ID: 8
F20 ID: 19
F26 ID: 25
F30 ID: 29
Core1 ID: 3
F10 ID: 9
F21 ID: 20
F23 ID: 22

F24 ID: 23
F25 ID: 24
F28 ID: 27
F29 ID: 28
Core2 ID: 4
F11 ID: 10
F22 ID: 21
F27 ID: 26
F31 ID: 30
Number of tasks :19
Group = 0:
0 1
Group = 3:
2 3 5 6
Group = 4:
4
Group = 7:
7 8
Group = 9:
9
Group = 10:
10
Group = 16:
11 12 16
Group = 13:
13
Group = 14:
14
Group = 15:
15
Group = 17:
17
Group = 18:
18
Group = 19:
19
Group = 23:
20 22 23
Group = 21:
21
Group = 24:
24 27
Group = 25:
25 29
Group = 30:
26 30
Group = 28:
28
Core3 ID: 0
Os application : ASIL_B
Os task: ASIL_B
runnable id: 18 runnable name : F19 runnable asil level : ASIL_B
Os application : ASIL_C
Os task: ASIL_C
runnable id: 0 runnable name : F1 runnable asil level : ASIL_C
runnable id: 1 runnable name : F2 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 15 runnable name : F16 runnable asil level : ASIL_D
Core4 ID: 1
Os application : ASIL_C
Os task: ASIL_C
runnable id: 2 runnable name : F3 runnable asil level : ASIL_C
runnable id: 3 runnable name : F4 runnable asil level : ASIL_C
runnable id: 5 runnable name : F6 runnable asil level : ASIL_C
runnable id: 6 runnable name : F7 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 11 runnable name : F12 runnable asil level : ASIL_D
runnable id: 12 runnable name : F13 runnable asil level : ASIL_D
runnable id: 16 runnable name : F17 runnable asil level : ASIL_D
I/O Core2 ID: 2
Os application : ASIL_B
Os task: ASIL_B
runnable id: 17 runnable name : F18 runnable asil level : ASIL_B
Os application : ASIL_C
Os task: ASIL_C
runnable id: 4 runnable name : F5 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 13 runnable name : F14 runnable asil level : ASIL_D
Os task: ASIL_D
runnable id: 14 runnable name : F15 runnable asil level : ASIL_D
Core1 ID: 3

```
Os application : ASIL_A
Os task: ASIL_A
runnable id: 24 runnable name : F25 runnable asil level : ASIL_A
runnable id: 27 runnable name : F28 runnable asil level : ASIL_A
Os task: ASIL_A
runnable id: 28 runnable name : F29 runnable asil level : ASIL_A Os application : ASIL_C
Os task: ASIL_C
runnable id: 20 runnable name : F21 runnable asil level : ASIL_C
runnable id: 22 runnable name : F23 runnable asil level : ASIL_C
runnable id: 23 runnable name : F24 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 9 runnable name : F10 runnable asil level : ASIL_D
Core2 ID: 4
Os application : ASIL_A
Os task: ASIL_A
runnable id: 26 runnable name : F27 runnable asil level : ASIL_A
runnable id: 30 runnable name : F31 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 21 runnable name : F22 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 10 runnable name : F11 runnable asil level : ASIL_D
I/O Core1 ID: 5
Os application : ASIL_A
Os task: ASIL_A
runnable id: 25 runnable name : F26 runnable asil level : ASIL_A
runnable id: 29 runnable name : F30 runnable asil level : ASIL_A
Os application : ASIL_C
Os task: ASIL_C
runnable id: 19 runnable name : F20 runnable asil level : ASIL_C
Os application : ASIL_D
Os task: ASIL_D
runnable id: 7 runnable name : F8 runnable asil level : ASIL_D
runnable id: 8 runnable name : F9 runnable asil level : ASIL_D
```

Figure 3.25: Solution for the automotive application and architecture model in the case of equal weights

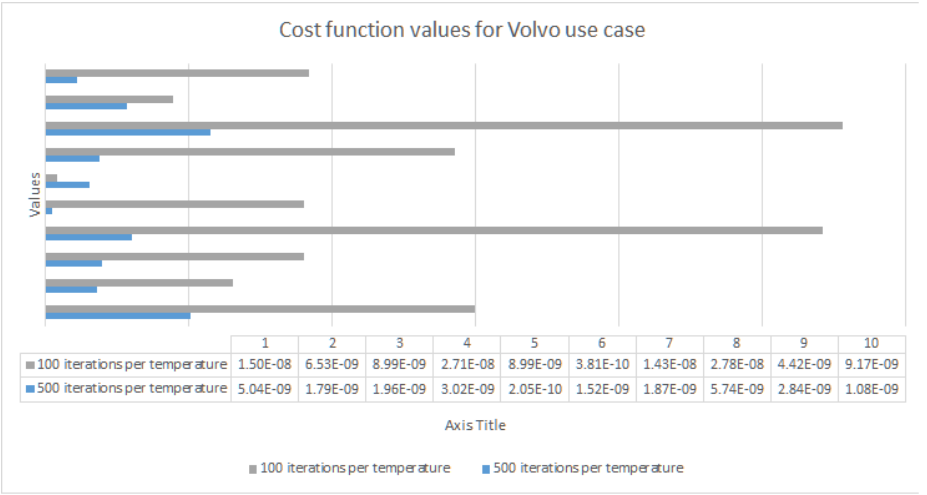
It is desirable to have a small number of *Os-Tasks* due to the overhead introduced by OS from context-switching from one task to another. In the cost function, if we only consider the communication bandwidth, the grouping of *Os-Tasks* does not affect the old cost value. After the transformation **S.3**, the new solution will have the same cost but the simulated annealing approach will still accept it. On the other hand, based on how the *runnables* are mapped into *Os-Tasks*, this might affect their $WCET_{communication}$, therefore the core utilization and the value of the cost function changes. For example, if we have two *runnables* exchanging signals that are mapped into the same *Os-Task*, their $WCET_{communication}$ will be added to the core utilization.

3.8.3 Volvo use case

For this use case, we ignore the overall inter-core bus utilization as Volvo was most interested in the core utilization. Nevertheless, the constraint that the inter-core bus utilization is not exceeded still remains. If the estimates for the $WCET_{communication}$ could not be obtained as it was the case with Volvo use case, the mapping tool can still work by setting $WCET_{communication}$ overheads to **0**.

As requested by Volvo, one additional constraint related to mapping of *runnables* to *Os-Tasks* has been added to allow *runnables* with the same period mapped into an *Os-Task*.

For this application model, we were interested in how the number of iterations per temperature influences the result of the simulated annealing. In fig. 3.26 different cost function values were obtained with the number of iterations per temperature being set to **100**, respectively to **500**. Overall, increasing the number of iterations per temperature allows simulated annealing approach to give better solutions (in this case with a smaller cost value).



	1	2	3	4	5	6	7	8	9	10
100 iterations per temperature	1.50E-08	6.53E-09	8.99E-09	2.71E-08	8.99E-09	3.81E-10	1.43E-08	2.78E-08	4.42E-09	9.17E-09
500 iterations per temperature	5.04E-09	1.79E-09	1.96E-09	3.02E-09	2.05E-10	1.52E-09	1.87E-09	5.74E-09	2.84E-09	1.08E-09

Figure 3.26: Cost function values for Volvo use case

The fig. 3.27 shows a solution that maps the *runnables* into the cores such that the core utilization is close to the overall mean. It is worth to notice that the solution almost has an evenly distributions of the *runnables* to cores: 24 (Core1), 25 (I/O core), 26 (Core2) and that all inter-core bus utilization are met.

```
Cost : 1.43554687500029E-09
total cpu is :1.43554687500029E-09
total bandwidth is :1.1520125
inter ecu communication is :0
inter core communication is :194
ECU1 ID: 0
c37 ID :0
c1455 ID :1
c1444 ID :2
c1447 ID :3
c1456 ID :4
c1425 ID :5
c1443 ID :6
c1453 ID :7
```

c1237 ID :8
c1440 ID :9
c1139 ID :10
c1477 ID :11
c1356 ID :12
c1171 ID :13
c1552 ID :14
c24 ID :15
c39 ID :16
c1331 ID :17
c1542 ID :18
c1545 ID :19
c1546 ID :20
c1417 ID :21
c1229 ID :22
c1400 ID :23
c1551 ID :24
c1256 ID :25
c57 ID :26
c1553 ID :27
c1454 ID :28
c1225 ID :29
c1358 ID :30
c1236 ID :31
c1374 ID :32
c1539 ID :33
c1233 ID :34
c1420 ID :35
c1458 ID :36
c1494 ID :37
c1462 ID :38
c63 ID :39
c1461 ID :40
c42 ID :41
c43 ID :42
c1479 ID :43
c1366 ID :44
c1550 ID :45
c1230 ID :46
c1534 ID :47
c1531 ID :48
c1452 ID :49
I/O Core1 ID: 2
a1650 ID: 0
a1671 ID: 2
a1661 ID: 4
a1699 ID: 7
a1669 ID: 8
a1654 ID: 11
a1711 ID: 13
a1658 ID: 15
a1655 ID: 16
a1665 ID: 17
a1643 ID: 21
a1646 ID: 24
a1641 ID: 26
a1678 ID: 27
a1708 ID: 28
a1636 ID: 39
a1638 ID: 42
a1712 ID: 43
a1696 ID: 45
a1660 ID: 46
a1703 ID: 47
a1694 ID: 55
a1653 ID: 60
a1637 ID: 61
a1633 ID: 62
Core2 ID: 1
a1630 ID: 1
a1667 ID: 3
a1645 ID: 19
a1635 ID: 20
a1651 ID: 23
a1713 ID: 25
a1685 ID: 29
a1693 ID: 30
a1702 ID: 32
a1704 ID: 33
a1639 ID: 35
a1664 ID: 36
a1706 ID: 40
a1697 ID: 44
a1707 ID: 51

a1683 ID: 54
a1673 ID: 56
a1701 ID: 57
a1632 ID: 59
a1674 ID: 63
a1688 ID: 64
a1631 ID: 66
a1652 ID: 67
a1700 ID: 68
a1677 ID: 72
a1668 ID: 74
Core1 ID: 0
a1672 ID: 5
a1666 ID: 6
a1691 ID: 9
a1698 ID: 10
a1689 ID: 12
a1676 ID: 14
a1710 ID: 18
a1687 ID: 22
a1657 ID: 31
a1695 ID: 34
a1663 ID: 37
a1692 ID: 38
a1670 ID: 41
a1681 ID: 48
a1682 ID: 49
a1656 ID: 50
a1690 ID: 52
a1640 ID: 53
a1675 ID: 58
a1648 ID: 65
a1659 ID: 69
a1662 ID: 70
a1679 ID: 71
a1705 ID: 73
Number of tasks :33
Group = 0:
0
Group = 1:
1 3 20 29 30 36 44 56 57 59 64 68 72 74
Group = 4:
2 4 11 13 16 17 21 24 26 27 39 46 47 55 60 61 62
Group = 5:
5
Group = 22:
6 9 14 18 22 31 38 41 58 69 73
Group = 7:
7
Group = 8:
8
Group = 10:
10
Group = 12:
12
Group = 15:
15
Group = 19:
19
Group = 23:
23
Group = 25:
25
Group = 28:
28
Group = 35:
32 35
Group = 33:
33
Group = 34:
34
Group = 37:
37
Group = 40:
40
Group = 42:
42
Group = 43:
43
Group = 45:
45
Group = 48:
48
Group = 49:

```
49
Group = 50:
50 52 65
Group = 51:
51
Group = 53:
53
Group = 54:
54
Group = 63:
63
Group = 66:
66
Group = 67:
67
Group = 70:
70
Group = 71:
71
Core1 ID: 0
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 5 runnable name : a1672 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 6 runnable name : a1666 runnable asil level : NO_ASIL
runnable id: 9 runnable name : a1691 runnable asil level : NO_ASIL
runnable id: 14 runnable name : a1676 runnable asil level : NO_ASIL
runnable id: 18 runnable name : a1710 runnable asil level : NO_ASIL
runnable id: 22 runnable name : a1687 runnable asil level : NO_ASIL
runnable id: 31 runnable name : a1657 runnable asil level : NO_ASIL
runnable id: 38 runnable name : a1692 runnable asil level : NO_ASIL
runnable id: 41 runnable name : a1670 runnable asil level : NO_ASIL
runnable id: 58 runnable name : a1675 runnable asil level : NO_ASIL
runnable id: 69 runnable name : a1659 runnable asil level : NO_ASIL
runnable id: 73 runnable name : a1705 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 10 runnable name : a1698 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 12 runnable name : a1689 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 34 runnable name : a1695 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 37 runnable name : a1663 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 48 runnable name : a1681 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 49 runnable name : a1682 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 50 runnable name : a1656 runnable asil level : NO_ASIL
runnable id: 52 runnable name : a1690 runnable asil level : NO_ASIL
runnable id: 65 runnable name : a1648 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 53 runnable name : a1640 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 70 runnable name : a1662 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 71 runnable name : a1679 runnable asil level : NO_ASIL
Core2 ID: 1
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 1 runnable name : a1630 runnable asil level : NO_ASIL
runnable id: 3 runnable name : a1667 runnable asil level : NO_ASIL
runnable id: 20 runnable name : a1635 runnable asil level : NO_ASIL
runnable id: 29 runnable name : a1685 runnable asil level : NO_ASIL
runnable id: 30 runnable name : a1693 runnable asil level : NO_ASIL
runnable id: 36 runnable name : a1664 runnable asil level : NO_ASIL
runnable id: 44 runnable name : a1697 runnable asil level : NO_ASIL
runnable id: 56 runnable name : a1673 runnable asil level : NO_ASIL
runnable id: 57 runnable name : a1701 runnable asil level : NO_ASIL
runnable id: 59 runnable name : a1632 runnable asil level : NO_ASIL
runnable id: 64 runnable name : a1688 runnable asil level : NO_ASIL
runnable id: 68 runnable name : a1700 runnable asil level : NO_ASIL
runnable id: 72 runnable name : a1677 runnable asil level : NO_ASIL
runnable id: 74 runnable name : a1668 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 19 runnable name : a1645 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 23 runnable name : a1651 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 25 runnable name : a1713 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 32 runnable name : a1702 runnable asil level : NO_ASIL
runnable id: 35 runnable name : a1639 runnable asil level : NO_ASIL
Os task: NO_ASIL
```

runnable id: 33 runnable name : a1704 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 40 runnable name : a1706 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 51 runnable name : a1707 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 54 runnable name : a1683 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 63 runnable name : a1674 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 66 runnable name : a1631 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 67 runnable name : a1652 runnable asil level : NO_ASIL
I/O Core1 ID: 2
Os application : NO_ASIL
Os task: NO_ASIL
runnable id: 0 runnable name : a1650 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 2 runnable name : a1671 runnable asil level : NO_ASIL
runnable id: 4 runnable name : a1661 runnable asil level : NO_ASIL
runnable id: 11 runnable name : a1654 runnable asil level : NO_ASIL
runnable id: 13 runnable name : a1711 runnable asil level : NO_ASIL
runnable id: 16 runnable name : a1655 runnable asil level : NO_ASIL
runnable id: 17 runnable name : a1665 runnable asil level : NO_ASIL
runnable id: 21 runnable name : a1643 runnable asil level : NO_ASIL
runnable id: 24 runnable name : a1646 runnable asil level : NO_ASIL
runnable id: 26 runnable name : a1641 runnable asil level : NO_ASIL
runnable id: 27 runnable name : a1678 runnable asil level : NO_ASIL
runnable id: 39 runnable name : a1636 runnable asil level : NO_ASIL
runnable id: 46 runnable name : a1660 runnable asil level : NO_ASIL
runnable id: 47 runnable name : a1703 runnable asil level : NO_ASIL
runnable id: 55 runnable name : a1694 runnable asil level : NO_ASIL
runnable id: 60 runnable name : a1653 runnable asil level : NO_ASIL
runnable id: 61 runnable name : a1637 runnable asil level : NO_ASIL
runnable id: 62 runnable name : a1633 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 7 runnable name : a1699 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 8 runnable name : a1669 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 15 runnable name : a1658 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 28 runnable name : a1708 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 42 runnable name : a1638 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 43 runnable name : a1712 runnable asil level : NO_ASIL
Os task: NO_ASIL
runnable id: 45 runnable name : a1696 runnable asil level : NO_ASIL

Figure 3.27: Solution for Volvo application model

Chapter 4

Conclusions and future work

This chapter presents the conclusions of the thesis and possible future work.

4.1 Conclusions

In this thesis, a method and a tool has been proposed for the problem of mapping AUTOSAR functionalities (*runnables*) with different ASIL levels on a distributed network of multi-core ECUs. An application model based on the AUTOSAR was defined in section 2.1. In addition, our solution is based on a scheduling, partitioning and communication model derived from the AUTOSAR specification (section 2.3).

The goal of the tool is to provide solutions that minimize the overall bus bandwidth and the variance of the core utilization while the safety constraints and the schedulability on each core are fulfilled.

Finding the optimal solution such that all the constraints are met is a NP-hard problem. This types of problems are in general impossible to solved by exploring the entire solution space, therefore it pays off to use algorithms that can return a solution close to the optimum. For our mapping problem, we have chosen the simulated annealing approach (section 3.5) together with a cost function in eq. (3.6). In addition, a backtracking algorithm has been implemented to test the quality of the solution obtained via simulated annealing.

Three use cases, each composed of an application and architecture model were defined to test the implementation (section 3.7). One use case has been proposed by Volvo Advanced Technology & Research in Götheborg, since they are interested in solutions for mapping functionalities on a multicore-ECU.

We have tested the implementation for different weights of the cost function's parameters and the experimental results were discussed in section 3.8.

4.2 Future work

Since in the automotive industry, applications often have to meet end-to-end timing constraints, defining new rules such that the tool provides a mapping solution were all the end-to-end timing constraints are met will be an important addition to the current implementation.

The authors in [LLP+09] have proposed new rules for mapping *runnables* to *Os-tasks* in AUTOSAR such that it minimizes the intra-ECU communication. The tool can be improved by adding such new rules and checking if we can obtain better mapping solutions given the new constraints.

Appendix A

Appendix

A.1 Volvo application model file


```
{ "runnableCommunication": [ [ "a1630", "a1703", 8 ], [ "a1653", "a1694", 1 ],
[ "a1658", "a1676", 16 ], [ "a1636", "a1704", 4 ],[ "a1712", "a1654", 5 ],
[ "a1693", "a1664", 8 ], [ "a1636", "a1663", 4 ], [ "a1655", "a1665", 10 ],
[ "a1693", "a1705", 16 ], [ "a1675", "a1666", 4 ], [ "a1673", "a1660", 16 ],
[ "a1694", "a1703", 4 ], [ "a1676", "a1692", 16 ], [ "a1653", "a1666", 41 ],
[ "a1675", "a1660", 4 ], [ "a1698", "a1632", 2 ], [ "a1694", "a1632", 12 ],
[ "a1675", "a1694", 4 ], [ "a1664", "a1662", 4 ],[ "a1657", "a1632", 4 ],
[ "a1635", "a1660", 9 ], [ "a1677", "a1698", 4 ], [ "a1711", "a1654", 6 ],
[ "a1635", "a1655", 8 ],[ "a1653", "a1702", 10 ], [ "a1653", "a1704", 18 ],
 [ "a1677", "a1632", 13 ], [ "a1703", "a1698", 4 ], [ "a1652", "a1632", 32 ],
[ "a1704", "a1655", 13 ], [ "a1636", "a1641", 1 ], [ "a1694", "a1695", 1 ],
[ "a1653", "a1676", 8 ], [ "a1702", "a1655", 13 ],[ "a1664", "a1665", 4 ],
[ "a1703", "a1676", 1 ], [ "a1692", "a1695", 184 ], [ "a1675", "a1691", 6 ],
[ "a1663", "a1660", 15 ],[ "a1653", "a1668", 148 ], [ "a1663", "a1655", 13 ],
[ "a1671", "a1660", 4 ], [ "a1711", "a1664", 4 ], [ "a1692", "a1632", 32 ],
[ "a1692", "a1703", 128 ], [ "a1697", "a1632", 9 ], [ "a1636", "a1654", 4 ],
[ "a1657", "a1711", 4 ], [ "a1653", "a1664", 16 ],[ "a1653", "a1662", 10 ],
[ "a1653", "a1672", 89 ], [ "a1664", "a1700", 4 ], [ "a1653", "a1697", 40 ],
[ "a1676", "a1675", 4 ], [ "a1662", "a1655", 13 ], [ "a1658", "a1672", 2 ],
[ "a1675", "a1698", 8 ], [ "a1692", "a1676", 128 ], [ "a1653", "a1639", 4 ],
[ "a1657", "a1662", 4 ], [ "a1654", "a1655", 36 ], [ "a1630", "a1669", 8 ],
[ "a1657", "a1663", 4 ], [ "a1655", "a1662", 10 ], [ "a1653", "a1632", 67 ],
[ "a1630", "a1676", 8 ], [ "a1675", "a1667", 9 ], [ "a1665", "a1655", 13 ],
[ "a1691", "a1692", 16 ],[ "a1653", "a1659", 25 ], [ "a1657", "a1687", 4 ],
[ "a1645", "a1702", 4 ], [ "a1653", "a1700", 51 ], [ "a1658", "a1660", 6 ],
[ "a1653", "a1703", 38 ], [ "a1703", "a1675", 3 ], [ "a1653", "a1671", 4 ],
[ "a1664", "a1705", 4 ], [ "a1635", "a1706", 8 ],[ "a1705", "a1701", 1 ],
```

["a1657", "a1641", 4], ["a1705", "a1694", 5], ["a1657", "a1657", 6],
["a1659", "a1666", 8],["a1674", "a1660", 2], ["a1658", "a1701", 4],
["a1630", "a1670", 8], ["a1655", "a1663", 10], ["a1695", "a1659", 16],
["a1692", "a1661", 1408], ["a1702", "a1675", 5], ["a1645", "a1662", 4],
["a1692", "a1660", 24], ["a1664", "a1704", 4],["a1658", "a1671", 2],
["a1693", "a1697", 16], ["a1675", "a1695", 8], ["a1658", "a1669", 2],
["a1693", "a1695", 44],["a1653", "a1663", 10], ["a1653", "a1692", 1200],
["a1703", "a1691", 34], ["a1656", "a1636", 1], ["a1630", "a1702", 8],
["a1630", "a1658", 8], ["a1677", "a1664", 1], ["a1653", "a1699", 4],
["a1653", "a1706", 198], ["a1693", "a1659", 16],["a1653", "a1698", 21],
["a1675", "a1659", 4], ["a1702", "a1676", 5], ["a1636", "a1705", 4],
["a1694", "a1664", 4],["a1693", "a1691", 8], ["a1692", "a1694", 32],
["a1657", "a1675", 4], ["a1653", "a1667", 56], ["a1693", "a1676", 4],
["a1630", "a1701", 8], ["a1630", "a1659", 8], ["a1697", "a1660", 4],
["a1693", "a1666", 16], ["a1636", "a1702", 4],["a1675", "a1697", 8],
["a1662", "a1660", 5], ["a1700", "a1701", 1], ["a1656", "a1655", 1],
["a1692", "a1693", 668],["a1657", "a1654", 5], ["a1705", "a1675", 5],
["a1630", "a1667", 8], ["a1693", "a1694", 29], ["a1706", "a1706", 14988],
["a1676", "a1702", 1], ["a1630", "a1661", 8], ["a1654", "a1712", 12],
["a1693", "a1661", 4], ["a1655", "a1655", 30],["a1693", "a1632", 44],
["a1655", "a1705", 10], ["a1657", "a1702", 4], ["a1674", "a1664", 1],
["a1658", "a1694", 4],["a1630", "a1672", 8], ["a1670", "a1667", 4],
["a1693", "a1675", 13], ["a1693", "a1699", 12], ["a1695", "a1664", 8],
["a1658", "a1668", 2], ["a1666", "a1675", 5], ["a1653", "a1711", 11],
["a1692", "a1691", 160], ["a1695", "a1632", 24],["a1657", "a1645", 4],
["a1701", "a1706", 60], ["a1666", "a1632", 20], ["a1658", "a1664", 2],
["a1630", "a1699", 8],["a1645", "a1665", 4], ["a1630", "a1705", 8],
["a1711", "a1703", 4], ["a1653", "a1660", 17], ["a1653", "a1654", 10],
["a1666", "a1677", 13], ["a1664", "a1702", 4], ["a1630", "a1700", 8],
["a1672", "a1703", 1], ["a1630", "a1693", 8],["a1630", "a1674", 8],
["a1700", "a1632", 1], ["a1635", "a1673", 16], ["a1660", "a1713", 8],
["a1658", "a1675", 2],["a1675", "a1701", 4], ["a1700", "a1655", 13],
["a1653", "a1658", 1], ["a1635", "a1701", 1], ["a1658", "a1703", 16],
["a1653", "a1705", 30], ["a1664", "a1632", 8], ["a1630", "a1695", 8],
["a1630", "a1694", 8], ["a1630", "a1698", 8], ["a1712", "a1632", 203],
["a1657", "a1665", 4], ["a1677", "a1666", 23], ["a1635", "a1667", 4],
["a1660", "a1655", 13],["a1657", "a1705", 4], ["a1655", "a1702", 10],
["a1711", "a1655", 10], ["a1664", "a1675", 4], ["a1630", "a1668", 8],
 ["a1630", "a1673", 8], ["a1664", "a1654", 4], ["a1693", "a1677", 4],
["a1694", "a1699", 4], ["a1658", "a1698", 2],["a1660", "a1632", 36],
["a1693", "a1701", 32],["a1645", "a1663", 4], ["a1655", "a1700", 10],
["a1699", "a1632", 14],["a1679", "a1679", 1], ["a1666", "a1697", 5],
["a1658", "a1667", 2], ["a1636", "a1665", 4], ["a1630", "a1697", 8],
 ["a1660", "a1701", 10], ["a1691", "a1675", 6], ["a1635", "a1636", 1],

```
[ "a1630", "a1691", 8 ], [ "a1658", "a1670", 2 ],[ "a1665", "a1660", 1 ],
[ "a1645", "a1705", 4 ], [ "a1675", "a1676", 9 ], [ "a1653", "a1674", 1 ],
[ "a1636", "a1700", 4 ],[ "a1658", "a1661", 16 ], [ "a1655", "a1636", 1 ],
[ "a1691", "a1697", 16 ], [ "a1630", "a1656", 8 ], [ "a1664", "a1679", 4 ],
[ "a1674", "a1669", 1 ], [ "a1705", "a1655", 13 ], [ "a1658", "a1666", 2 ],
[ "a1659", "a1703", 8 ], [ "a1653", "a1669", 152 ],[ "a1630", "a1677", 8 ],
[ "a1657", "a1700", 4 ], [ "a1664", "a1663", 4 ], [ "a1679", "a1654", 5 ],
[ "a1675", "a1703", 8 ],[ "a1659", "a1632", 16 ], [ "a1630", "a1671", 8 ],
[ "a1653", "a1665", 10 ], [ "a1657", "a1635", 4 ], [ "a1675", "a1632", 42 ],
[ "a1688", "a1654", 5 ], [ "a1653", "a1677", 79 ], [ "a1657", "a1713", 4 ],
[ "a1630", "a1666", 8 ], [ "a1674", "a1661", 1 ],[ "a1692", "a1698", 32 ],
[ "a1630", "a1664", 8 ], [ "a1645", "a1700", 4 ], [ "a1655", "a1654", 4 ],
[ "a1653", "a1670", 4 ],[ "a1703", "a1632", 21 ], [ "a1658", "a1691", 20 ],
[ "a1698", "a1701", 2 ], [ "a1630", "a1663", 8 ], [ "a1674", "a1675", 2 ],
[ "a1697", "a1659", 8 ], [ "a1636", "a1662", 4 ], [ "a1675", "a1706", 4 ],
[ "a1630", "a1692", 8 ], [ "a1661", "a1664", 2 ],[ "a1674", "a1668", 1 ],
[ "a1673", "a1655", 8 ], [ "a1668", "a1664", 2 ], [ "a1697", "a1703", 4 ],
[ "a1660", "a1679", 1 ],[ "a1669", "a1664", 1 ], [ "a1630", "a1660", 8 ],
[ "a1630", "a1704", 8 ], [ "a1630", "a1665", 8 ], [ "a1674", "a1701", 1 ],
[ "a1700", "a1694", 5 ], [ "a1658", "a1677", 2 ], [ "a1693", "a1700", 8 ],
[ "a1630", "a1675", 8 ], [ "a1630", "a1662", 8 ], [ "a1653", "a1661", 1 ],
[ "a1653", "a1695", 17 ], [ "a1670", "a1660", 4 ], [ "a1645", "a1635", 4 ],
[ "a1691", "a1698", 16 ],[ "a1693", "a1703", 17 ], [ "a1664", "a1697", 1 ],
[ "a1675", "a1664", 4 ], [ "a1636", "a1656", 1 ], [ "a1695", "a1666", 4 ],
[ "a1672", "a1664", 1 ], [ "a1687", "a1673", 1 ], [ "a1711", "a1656", 4 ],
[ "a1655", "a1704", 10 ], [ "a1687", "a1654", 5 ], [ "a1653", "a1675", 52 ],
[ "a1660", "a1675", 1 ], [ "a1653", "a1655", 340 ], [ "a1696", "a1654", 5 ],
[ "a1660", "a1706", 12 ], [ "a1654", "a1711", 4 ], [ "a1658", "a1693", 20 ],
[ "a1653", "a1701", 98 ], [ "a1645", "a1704", 4 ], [ "a1703", "a1697", 2 ],
[ "a1657", "a1704", 4 ], [ "a1653", "a1652", 12 ], [ "a1658", "a1695", 20 ],
 [ "a1674", "a1698", 1 ] ],
 "softwareComponents": [ { "asil": "NO_ASIL", "name": "c37", "runnables":
 [ { "deadline": 10.0, "name": "a1650", "offset": 0.0, "period": 10.0,
 "wcet": 0.0015 },
 { "deadline": 1.25, "name": "a1630", "offset": 0.0, "period": 1.25,
 "wcet": 0.003 } ] },
 { "asil": "NO_ASIL", "name": "c1455", "runnables":
 [ { "deadline": 10.0, "name": "a1671", "offset": 0.0, "period": 10.0,
 "wcet": 0.0165 } ] },
 { "asil": "NO_ASIL", "name": "c1444", "runnables":
 [ { "deadline": 10.0, "name": "a1667", "offset": 0.0, "period": 10.0,
 "wcet": 0.0285 } ] },
 { "asil": "NO_ASIL", "name": "c1447", "runnables":
 [ { "deadline": 10.0, "name": "a1661", "offset": 0.0, "period": 10.0,
```

```
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c1456", "runnables":
[ { "deadline": 10.0, "name": "a1672", "offset": 0.0, "period": 10.0,
"wcet": 0.010499999999999999 } ] },
{ "asil": "NO_ASIL", "name": "c1425", "runnables":
[ { "deadline": 10.0, "name": "a1666", "offset": 0.0, "period": 10.0,
"wcet": 0.1005 } ] },
{ "asil": "NO_ASIL", "name": "c1443", "runnables":
[ { "deadline": 20.0, "name": "a1699", "offset": 0.0, "period": 20.0,
"wcet": 0.015000000000000003 } ] },
{ "asil": "NO_ASIL", "name": "c1453", "runnables":
[ { "deadline": 10.0, "name": "a1669", "offset": 0.0, "period": 10.0,
"wcet": 0.045000000000000005 } ] },
{ "asil": "NO_ASIL", "name": "c1237", "runnables":
[ { "deadline": 20.0, "name": "a1691", "offset": 0.0, "period": 20.0,
"wcet": 0.051 } ] },
{ "asil": "NO_ASIL", "name": "c1440", "runnables":
[ { "deadline": 20.0, "name": "a1698", "offset": 0.0, "period": 20.0,
"wcet": 0.0165 } ] },
{ "asil": "NO_ASIL", "name": "c1139", "runnables":
[ { "deadline": 10.0, "name": "a1654", "offset": 0.0, "period": 10.0,
"wcet": 0.022500000000000003 },
{ "deadline": 20.0, "name": "a1689", "offset": 0.0, "period": 20.0,
"wcet": 0.0015 },
{ "deadline": 160.0, "name": "a1711", "offset": 0.0, "period": 160.0,
"wcet": 0.312 } ] },
{ "asil": "NO_ASIL", "name": "c1477", "runnables":
[ { "deadline": 10.0, "name": "a1676", "offset": 0.0, "period": 10.0,
"wcet": 0.034499999999999996 } ] },
{ "asil": "NO_ASIL", "name": "c1356", "runnables":
[ { "deadline": 10.0, "name": "a1658", "offset": 0.0, "period": 10.0,
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c1171", "runnables":
[ { "deadline": 10.0, "name": "a1655", "offset": 0.0, "period": 10.0,
"wcet": 0.0015 } ] },
{ "asil": "NO_ASIL", "name": "c1552", "runnables":
[ { "deadline": 10.0, "name": "a1665", "offset": 0.0, "period": 10.0,
"wcet": 0.0285 } ] },
{ "asil": "NO_ASIL", "name": "c24", "runnables":
[ { "deadline": 160.0, "name": "a1710", "offset": 0.0, "period": 160.0,
"wcet": 0.0015 },
{ "deadline": 5.0, "name": "a1645", "offset": 0.0, "period": 5.0,
"wcet": 0.0135 },
{ "deadline": 80.0, "name": "a1635", "offset": 0.0, "period": 80.0,
```

```
"wcet": 0.039 },
{ "deadline": 0.15, "name": "a1643", "offset": 0.0, "period": 0.15,
"wcet": 0.0 },
{ "deadline": 20.0, "name": "a1687", "offset": 0.0, "period": 20.0,
"wcet": 0.0075000000000000015 },
{ "deadline": 10.0, "name": "a1651", "offset": 0.0, "period": 10.0,
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c39", "runnables":
[ { "deadline": 5.0, "name": "a1646", "offset": 0.0, "period": 5.0,
"wcet": 0.006 },
{ "deadline": 160.0, "name": "a1713", "offset": 0.0, "period": 160.0,
"wcet": 0.003 },
{ "deadline": 80.0, "name": "a1641", "offset": 0.0, "period": 80.0,
"wcet": 0.003 },
{ "deadline": 10.0, "name": "a1678", "offset": 0.0, "period": 10.0,
"wcet": 0.0 },
{ "deadline": 20.0, "name": "a1708", "offset": 0.0, "period": 20.0,
"wcet": 0.0015 },
{ "deadline": 40.0, "name": "a1685", "offset": 0.0, "period": 40.0,
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c1331", "runnables":
[ { "deadline": 20.0, "name": "a1693", "offset": 0.0, "period": 20.0,
"wcet": 0.10650000000000001 } ] },
{ "asil": "NO_ASIL", "name": "c1542", "runnables":
[ { "deadline": 10.0, "name": "a1657", "offset": 0.0, "period": 10.0,
"wcet": 0.0015 } ] },
{ "asil": "NO_ASIL", "name": "c1545", "runnables":
[ { "deadline": 20.0, "name": "a1702", "offset": 0.0, "period": 20.0,
"wcet": 0.034499999999999996 } ] },
{ "asil": "NO_ASIL", "name": "c1546", "runnables":
[ { "deadline": 20.0, "name": "a1704", "offset": 0.0, "period": 20.0,
"wcet": 0.033 } ] },
{ "asil": "NO_ASIL", "name": "c1417", "runnables":
[ { "deadline": 20.0, "name": "a1695", "offset": 0.0, "period": 20.0,
"wcet": 0.102 } ] },
{ "asil": "NO_ASIL", "name": "c1229", "runnables":
[ { "deadline": 80.0, "name": "a1639", "offset": 0.0, "period": 80.0,
"wcet": 0.0015 } ] },
{ "asil": "NO_ASIL", "name": "c1400", "runnables":
[ { "deadline": 10.0, "name": "a1664", "offset": 0.0, "period": 10.0,
"wcet": 0.0435 } ] },
{ "asil": "NO_ASIL", "name": "c1551", "runnables":
[ { "deadline": 10.0, "name": "a1663", "offset": 0.0, "period": 10.0,
"wcet": 0.041999999999999996 } ] },
```

```
{ "asil": "NO_ASIL", "name": "c1256", "runnables":
[ { "deadline": 20.0, "name": "a1692", "offset": 0.0, "period": 20.0,
"wcet": 0.22349999999999998 } ] },
{ "asil": "NO_ASIL", "name": "c57", "runnables":
[ { "deadline": 80.0, "name": "a1636", "offset": 0.0, "period": 80.0,
"wcet": 0.0045000000000000005 } ] },
{ "asil": "NO_ASIL", "name": "c1553", "runnables":
[ { "deadline": 20.0, "name": "a1706", "offset": 0.0, "period": 20.0,
"wcet": 0.010499999999999999 } ] },
{ "asil": "NO_ASIL", "name": "c1454", "runnables":
[ { "deadline": 10.0, "name": "a1670", "offset": 0.0, "period": 10.0,
"wcet": 0.0165 } ] },
{ "asil": "NO_ASIL", "name": "c1225", "runnables":
[ { "deadline": 80.0, "name": "a1638", "offset": 0.0, "period": 80.0,
"wcet": 0.0015 },
{ "deadline": 160.0, "name": "a1712", "offset": 0.0, "period": 160.0,
"wcet": 0.003 } ] },
{ "asil": "NO_ASIL", "name": "c1358", "runnables":
[ { "deadline": 20.0, "name": "a1697", "offset": 0.0, "period": 20.0,
"wcet": 0.041999999999999996 } ] },
{ "asil": "NO_ASIL", "name": "c1236", "runnables":
[ { "deadline": 20.0, "name": "a1696", "offset": 0.0, "period": 20.0,
"wcet": 0.027 } ] },
{ "asil": "NO_ASIL", "name": "c1374", "runnables":
[ { "deadline": 10.0, "name": "a1660", "offset": 0.0, "period": 10.0,
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c1539", "runnables":
[ { "deadline": 20.0, "name": "a1703", "offset": 0.0, "period": 20.0,
"wcet": 0.063 } ] },
{ "asil": "NO_ASIL", "name": "c1233", "runnables":
[ { "deadline": 320.0, "name": "a1681", "offset":
0.0, "period": 320.0, "wcet": 0.003 },
{ "deadline": 320.0, "name": "a1682", "offset":
0.0, "period": 320.0, "wcet": 0.0 },
{ "deadline": 10.0, "name": "a1656", "offset": 0.0, "period": 10.0,
"wcet": 0.009000000000000001 },
{ "deadline": 20.0, "name": "a1707", "offset":
0.0, "period": 20.0, "wcet": 0.0 },
{ "deadline": 20.0, "name": "a1690", "offset": 0.0, "period": 20.0,
"wcet": 0.003 },
{ "deadline": 80.0, "name": "a1640", "offset": 0.0, "period": 80.0,
"wcet": 0.0015 },
{ "deadline": 320.0, "name": "a1683", "offset": 0.0, "period": 320.0,
"wcet": 0.003 } ] },
```

```
{ "asil": "NO_ASIL", "name": "c1420", "runnables":
[ { "deadline": 20.0, "name": "a1694", "offset": 0.0, "period": 20.0,
"wcet": 0.0285 } ] },
{ "asil": "NO_ASIL", "name": "c1458", "runnables":
[ { "deadline": 10.0, "name": "a1673", "offset": 0.0, "period": 10.0,
"wcet": 0.022500000000000003 } ] },
{ "asil": "NO_ASIL", "name": "c1494", "runnables":
[ { "deadline": 20.0, "name": "a1701", "offset": 0.0, "period": 20.0,
"wcet": 0.099 } ] },
{ "asil": "NO_ASIL", "name": "c1462", "runnables":
[ { "deadline": 10.0, "name": "a1675", "offset": 0.0, "period": 10.0,
"wcet": 0.0795 } ] },
{ "asil": "NO_ASIL", "name": "c63", "runnables":
[ { "deadline": 1.25, "name": "a1632", "offset": 0.0, "period": 1.25,
"wcet": 0.012 },
{ "deadline": 10.0, "name": "a1653", "offset": 0.0, "period": 10.0,
"wcet": 0.13349999999999998 },
{ "deadline": 80.0, "name": "a1637", "offset": 0.0, "period": 80.0,
"wcet": 0.0165 },
{ "deadline": 1.25, "name": "a1633", "offset": 0.0, "period": 1.25,
"wcet": 0.027 } ] },
{ "asil": "NO_ASIL", "name": "c1461", "runnables":
[ { "deadline": 10.0, "name": "a1674", "offset": 0.0, "period": 10.0,
"wcet": 0.006 } ] },
{ "asil": "NO_ASIL", "name": "c42", "runnables":
[ { "deadline": 20.0, "name": "a1688", "offset": 0.0, "period": 20.0,
"wcet": 0.0015 } ] },
{ "asil": "NO_ASIL", "name": "c43", "runnables":
[ { "deadline": 2.5, "name": "a1648", "offset": 0.0, "period": 2.5,
"wcet": 0.0 },
{ "deadline": 1.25, "name": "a1631", "offset": 0.0, "period": 1.25,
"wcet": 0.003 },
{ "deadline": 10.0, "name": "a1652", "offset": 0.0, "period": 10.0,
"wcet": 0.006 } ] },
{ "asil": "NO_ASIL", "name": "c1479", "runnables":
[ { "deadline": 20.0, "name": "a1700", "offset": 0.0, "period": 20.0,
"wcet": 0.058499999999999996 } ] },
{ "asil": "NO_ASIL", "name": "c1366", "runnables":
[ { "deadline": 10.0, "name": "a1659", "offset": 0.0, "period": 10.0,
"wcet": 0.0 } ] },
{ "asil": "NO_ASIL", "name": "c1550", "runnables":
[ { "deadline": 10.0, "name": "a1662", "offset": 0.0, "period": 10.0,
"wcet": 0.030000000000000006 } ] },
{ "asil": "NO_ASIL", "name": "c1230", "runnables":
```

```
[ { "deadline": 10.0, "name": "a1679", "offset": 0.0, "period": 10.0,
"wcet": 0.0165 } ] },
{ "asil": "NO_ASIL", "name": "c1534", "runnables":
[ { "deadline": 10.0, "name": "a1677", "offset": 0.0, "period": 10.0,
"wcet": 0.04650000000000001 } ] },
{ "asil": "NO_ASIL", "name": "c1531", "runnables":
[ { "deadline": 20.0, "name": "a1705", "offset": 0.0, "period": 20.0,
"wcet": 0.04650000000000001 } ] },
{ "asil": "NO_ASIL", "name": "c1452", "runnables":
[ { "deadline": 10.0, "name": "a1668", "offset": 0.0, "period": 10.0,
"wcet": 0.045000000000000005 } ] } ] }
```

Bibliography

[ATPK+11] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gerard, and F. Terrier. Enabling scheduling analysis for autosar systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 152–159, March 2011.

[AUTa] ERIKA AUTOSAR. *ERIKA AUTOSAR OS-Application example.* `http://erika.tuxfamily.org/wiki/index.php?title=Erika_AUTOSAR_OS`.

[AUTb] ERIKA AUTOSAR. *ERIKA AUTOSAR Timing Protection example.* `http://erika.tuxfamily.org/wiki/index.php?title=Erika_AUTOSAR_OS#Timing_Protection`.

[AUT14a] AUTOSAR_EXP_VFB. Virtual functional bus. Technical report, AUTOSAR 4.2.1, 2014.

[AUT14b] AUTOSAR_SW_OS. Specification of operating system. Technical report, AUTOSAR 4.2.1, 2014.

[AUT14c] AUTOSAR_SWS_RTE. Specification of rte. Technical report, AUTOSAR 4.2.1, 2014.

[AUT14d] AUTOSAR_TR_SafetyConceptStatusReport. Technical safety concept status report. Technical report, AUTOSAR 4.2.1, 2014.

[BFWS10] S. Bunzel, S. Furst, J. Wagenhuber, and F. Stappert. *Safety and security related features in autosar*, 2010. `http://www.automotive2010.de/programm/contentdata/Bunzel-AUTOSAR.pdf`.

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[CDKM02] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems.* Wiley, 2002.

[FFR12] Christoph Ficek, Nico Feiertag, and Dr. Kai Richter. *Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems*, 2012. http://web1.see.asso.fr/erts2012/Site/0P2RUC89/4C-4.pdf.

[FLSN14] H.R. Faragardi, B. Lisper, K. Sandstrom, and T. Nolte. An efficient scheduling of autosar runnables to minimize communication cost in multi-core systems. In *Telecommunications (IST), 2014 7th International Symposium on*, pages 41–48, Sept 2014.

[Fre11] Patrick Frey. *A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems*, 2011. http://vts.uni-ulm.de/query/longview.meta.asp?document_id=7505.

[GHAG11] Peter Gliwa (Gliwa GmbH), Jens Harnisch, Ursula Kelling (Infineon Technologies AG), and Christoph Ficek (SYMTAVISION GmbH). *From Single-Core to Multi-Core Platforms, Systematic Migration of Hard Real-Time Software in AUTOSAR*, 2011. https://www.symtavision.com/downloads/success-stories/From_Single-Core_to_Multi-Core_Platforms_Symtavision_Infineon_Gliwa_Embedded_World_2011_presentation.pdf.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gli13] Peter Gliwa. *Timing Analysis - Poster*, February 2013. "http://www.gliwa.com/downloads/Timing%20Poster.pdf".

[GUL] Pascal GULA. *Semantic of execution in AUTOSAR*. http://www.telecom-paristech.fr/ETR09/presenta/Gula.pdf.

[HC08] HYUNDAI MOTOR Company HYUNDAI and KPIT Cummins. Performance of autosar basic software modules in a chassis ecu, 2008.

[IPEP06] V. Izosimov, P. Pop, P. Eles, and Zebo Peng. Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[LLP+09] Rongshen Long, Hong Li, Wei Peng, Yi Zhang, and Minde Zhao. An approach to optimize intra-ecu communication based on mapping of autosar runnable entities. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 138–143, May 2009.

[Mat14] Alexander Mattausch. *Do AUTOSAR and functional safety rule each other out?*, 2014. `https://d23rjziej2pu9i.cloudfront.net/wp-content/uploads/2014/11/28025218/AUTOSAR_Functional_Safety_Do_they_rule_each_other_out.pdf`.

[NMBSL10] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. Multi-source and multicore automotive ecus - os protection mechanisms and scheduling. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pages 3734–3741, July 2010.

[PPEP08] Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. Analysis and optimisation of hierarchically scheduled multiprocessor embedded systems. *Int. J. Parallel Program.*, 36(1):37–67, February 2008.

[Ree93] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[SCCM15] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin Marteil. An ilp approach for mapping autosar runnables on multi-core architectures. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '15, pages 6:1–6:8, New York, NY, USA, 2015. ACM.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[SR08] O. Scheickl and M. Rudorfer. Automotive real time development using a timing-augmented autosar specification. *Proc. ERTS*, 2008.

[VDX05] OSEK VDX. *Specification OSEK OS 2.2.3 - OSEK/VDX*, 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller,

Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[WMM+13] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. An optimization approach for the synthesis of autosar architectures. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–10, Sept 2013.