(ITEA 2 – 13017)

Enabling of Results from AMALTHEA and others
for Transfer into Application and
building Community around

Deliverable: D 2.2

Concept for Partitioning, Mapping, and Tracing for Multi- and Many-core Systems

Work Package: 2
Systems engineering

Task: 2.2
Techniques for Partitioning, Mapping, and Tracing

Document Type:	Deliverable	**Classification:**	Public
Document Version:	draft	**Contract Start Date:**	01.09.2014
Document Preparation Date:	31.05.2016	**Duration:**	31.08.2017

ITEA 2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

EUREKA

History

Rev.	Content	Resp. Partner	Date
0.1	Various initial contributions	all partners	01.07.2016
0.2	Added content to 4.2 and 4.5	Krawczyk (DoUAS)	08.07.2016
0.3	Finished partitioning chapter	Höttger (DoUAS)	12.07.2016
0.4	Added content to resource management chapter	Höttger (DoUAS)	14.07.2016
0.5	Added content to resource management chapter and included information functional safety	Höttger (DoUAS)	22.07.2016
0.6	Corrected several mathematical notations in resource management chapter	Höttger (DoUAS)	27.07.2016
0.7	Reviewed safety considerations chapter	Höttger (DoUAS)	27.07.2016
0.8	Finished resource management chapter and reviewed mapping chapter	Höttger (DoUAS)	02.08.2016
0.9	Finished chapter 4 - Scheduling for ECU Networks	David Schmelter (Fraunhofer IEM)	04.08.2016
1.0	Final review	Höttger (DoUAS)	05.08.2016

Final Approval	Name	Partner
WP2 Internal	Grewing, Höttger, Johr, Krawczyk, Schmelter, Schmidt,	BHTC, DoUAS, FhG IEM, TWT
WP2 Leader	Kamsties	Dortmund University of Applied Sciences and Arts

Contents

List of Figures

List of Tables

Summary

The deliverable D2.2 describes the Workpackage 2 concept for partitioning, mapping, and tracing for multi- and many-core systems. This deliverable complements the previous deliverable D2.1 "Concept for Requirements and Architectural Models for Multicore Systems": D2.1 focuses on the front-end activities while D2.2 focuses on the back-end activities.

With respect to the back-end activities, the first part of this deliverable is devoted to *partitioning* and *mapping*. New methods and improved implementation of these tools are discussed to deal with feedback from industrial application.

The second part of this deliverable discusses a couple of aspects related to partitioning and mapping: an extended development process for partitioning and mapping of ECU networks, safety aspects in partitioning and mapping, management of shared resources, and further development of the WP2 case studies.

Acronyms

API	Application Programming Interface
ASL	Allocation Specification Language
CIC	Component Instance Configuration
ECU	Electronic Control Unit
ILP	Integer Linear Programming
ITS	Intelligent Technical System
OCL	Object Constraint Language
PDM	Platform Description Model
PIM	Platform Independent Model
PSM	Platform Specific Modeling
QoS	Quality of Service
RTCP	Real-time Coordination Protocol
RTSC	Real-time State Chart
WCET	Worst Case Execution Time
ASIL	Automotive Safety Integrity Level
WCRT	Worst Case Response Time

1 Introduction

The deliverable D2.2 describes the Workpackage 2 concept for partitioning, mapping, and tracing for multi- and many-core systems. This deliverable complements the previous deliverable D2.1 "Concept for Requirements and Architectural Models for Multicore Systems": D2.1 focuses on the front-end activities while D2.2 focuses on the back-end activities.

With respect to the back-end activities, the first part of this deliverable (Chapter 2 and 3) is devoted to *partitioning* and *mapping*. Partitioning was already a subject of the past Amalthea project. In Amalthea4public, we incorporate into the partitioning tool experiences we collected from applications in industrial practice in the meanwhile:

- More algorithms - we investigate so-called *list scheduling* and *bin packing algorithms* for partitioning (i.e., assigning runnables to tasks)
- Automation - Partitioning should be accessible by workflows written using the Eclipse Advanced Scripting Environment (EASE)
- Visualization - the results of partitioning call for a graphical representation to make them accessible to an embedded software engineer

Similar to partitioning, a mapping tool based on Integer Linear Programming (ILP) was developed in the past Amalthea project. This mapping tool is also enhanced in Amalthea4public according to new requirements and experiences made in industrial practice:

- More algorithms - a new approach to mapping based on genetic algorithms was investigated to address more than one quality attribute (e.g., performance *and* energy) and extensions to ILP
- More degrees of freedom - aspects that may change during optimization
- Constraints - mapping should be aware of various constraints that effectively reduce the solution space

The second part of this deliverable (Chapter 4-7) discusses a couple of aspects related to partitioning and mapping:

- extended development process for partitioning and mapping of ECU networks,
- safety aspects in partitioning and mapping,
- management of shared resources, and
- further development of the case studies introduced in Deliverable D2.1.

The intended readers are developers involved in embedded multicore development and tool providers who are looking for new techniques to add to their tooling.

2 Partitioning

Partitioning in App4mc is the initial approach to identify partitions respectively tasks that can run in parallel on different processing cores without violating various constraints, *e.g.*, in regard to timing, safety, behavior, *etc.* The partitioning feature is briefly introduced in the App4mc help documentation and further described in detail in Amalthea deliverable D3.4 [7]. In Amalthea4public, we plan to extend the partitioning approaches available in the App4mc platform and provide more effective and more efficient implementations. Apart from accessibility through *Javascript* workflows (*e.g.*, shown in the help documentation), we evaluated various existing applicable approaches that could improve efficiency or partitioning results for specific use cases. These approaches and methodologies are described in the following sections.

The remainder of this chapter is structured as follows: The next section 2.1 presents the idea of some applicable list scheduling approaches that can be utilized in App4mc by means of new partitioning strategies. Afterwards, section 2.2 describes some new features that affect efficiency and effectiveness of possible partitioning results. Section 2.3 describes how the partitioning approaches can be used and configured via a workflow to perform multiple generations among various models and different configurations sequentially in one single workflow. Afterwards, Section 2.4 describes and compares four different possibilities to visualize runnable dependency graphs, *i.e.*, the basis for the partitioning strategies. Finally, section 2.5 sums up the different sections and forms a conclusion of this partitioning chapter.

2.1 Related Partitioning Strategies

In order to keep this chapter simple, the state-of-the-art analysis is reduced to the strategies that might be applicable to Amalthea models in App4mc. Some promising list scheduling approaches are described in the following sections 2.1.2 and 2.1.1. App4mc's basic parallelization idea is based upon Foster's approach presented in [25].

The implemented approaches *ESSP* and *CPP* (in App4mc) focus on causal order based analysis of a software model's parallelization potential. While considering runnable instructions (node weights) and communication overheads (edge weights), a runnable to task distribution is identified to reduce overall response times. As soon as runnable instructions deviate from a specific constant and have a dynamic behavior, list scheduling approaches described in sections 2.1.1 and 2.1.2 can help to provide more effective results due to instructions being related to specific processes. Moreover, some applications may not require causal order consideration. In this case, bin packing algorithms can provide a more efficient way to get partitioned results (see Section 2.1.4).

2.1.1 HEFT

List scheduling algorithms perform their metric calculations based on a graph $G = \{V, E\}$ with $V = \{v_1, ..., v_n\}$ representing n vertexes (*i.e.*, runnables) and $E = \{e_1, ..., e_m\}$ (*i.e.*, runnable

dependencies: `RunnableSequencingConstraints`) representing m directed edges between two vertexes v_{source} and v_{target}. Vertexes and Edges have a value denoting its weight. List scheduling algorithms further have computation cost matrix denoting costs of a vertex on different processors.

Consequently, the *HEFT* algorithm is a list scheduling algorithm [48] for heterogeneous and dynamic software behavior (Heterogeneous Earliest-Finish-Time) [48]. Based on two metrics, the algorithm's goal is to minimize Signal-to-Length-Ratio (SLR). The first metric defines an upper rank:

$$rank_u(v_i) = \overline{w_i} + \max_{v_j \in succ(v_i)} (\overline{c_{i,j}} + rank_u(v_j)) \tag{2.1}$$

The second metric defines a downward rank:

$$rank_u(v_i) = \max_{v_j \in prec(v_i)} \{rank_d(v_j) + \overline{w_j} + \overline{c_{j,i}}\} \tag{2.2}$$

with w_i denoting a vertex's mean weight, *i.e.*, the mean runnable instructions and $\overline{c_{i,j}}$ denoting the communication cost between v_i and v_j. Further, the $rank_u$ of exit vertexes is denoted by their mean weight: $rank_u(v_{exit}) = \overline{w_{exit}}$. Finally, together with the computation cost matrix, the algorithm identifies the sum of the both ranks as the priority of a vertex to be scheduled next (the higher the value is, the earlier it is chosen for processor assignment). Processor assignment is then based on greedy minimization of the total SLR. In terms of partitioning, processor assignment can be considered as partition (`ProcessPrototype`) assignment. *HEFT* scheduling was the first to improve Levelized-Min Time (LMT).

2.1.2 PETS

The PETS algorithm (Performance Effective Task Scheduling) was introduced in [32] and can be used in App4mc since it comprises the consideration of heterogeneous and dynamic software behavior in form of a list scheduling algorithm. Instead of statically deriving execution times from processor speeds and function instructions, the PETS algorithm relies on a computation cost matrix $T \times P$ with $T = \{T_0, T_1, ...T_n\}$, n = number of tasks and $P = \{P_0, P_1, ...P_m\}$, m = number of processors. The overall optimization goal is to minimize the schedule length: $SL = \max\{\text{AFT}(v_{exit})\}$, with AFT = actual finish time and v_{exit} denoting all exit vertexes (*i.e.*, runnables that have no outdoing dependency). The algorithm performs level sorting, vertex prioritization and processor selection. Level sorting is done to process the given vertexes in topological order. The vertex prioritization assigns a value to each vertex (*i.e.*, its priority) regarding the vertex's average computation cost (ACC), the data transfer cost (DTC), and a rank of predecessor vertex (RPT). Where ACC and DTC are clear, RPT is defined via: $\text{RPT}(v_x) = \max(\text{rank}(v_w))$ where v_w are immediate predecessors of v_x and $\text{rank}_{v_i} = \text{ACC}_{v_i} + \text{DTC}_{v_i} + \text{RPT}_{v_i}$. A vertex's priority is a topological level based integer value starting with one (for the highest rank value), that increases by one for each vertex at the same topological level regarding the next lower rank value. The highest rank receives priority 1, the lowest rank receives the highest integer in its topological level. The priority value is used to decide in which order tasks are selected. In this order, the vertexes (v_i) are assigned to a processor that minimizes $\text{AFT}(v_i)$. The results are very effective for *FFT* graph structures but produces worse results for random graph generators with higher amounts of tasks [10].

2.1.3 PEFT

Arabnejad *et al.* assess many list scheduling algorithms regarding random task graphs, gaussian elimination approaches, fast fourier transformation, montage workflows and epigenomic workflows in [10]. They further present a new list scheduling algorithm Predict Earlist Finish Time (PEFT) that further introduces an optimistic cost table:

$$\mathrm{OCT}(v_i, p_k) = \max_{v_j \in \mathrm{succ}(v_i)} \left[\min_{p_w \in P} \{ \mathrm{OCT}(v_j, p_w) + w(v_j, p_w) + \overline{c_{i,j}} \} \right] \tag{2.3}$$

with $\overline{c_{i,j}} = 0 \quad if \quad p_w = p_k$, *i.e.*, the average communication cost $\overline{c_{i,j}}$ is 0 if v_i is evaluated for processor p_k. Similar to the previously described approaches, v_i would correspond a runnable, p_w a partition or task, and $\overline{c_{i,j}}$ the communication cost between runnables i and j. OCT is used in the task prioritizing phase to determine ranks. Instead of three phases, *PEFT* only performs task prioritization and processor selection. During processor (*i.e.*, partition) selection, a rank is calculated via *OCT*, that defines which task is selected to assignment :

$$\mathrm{rank}_{\mathrm{OCT}}(v_i) = \frac{\sum_{k=1}^{P} \mathrm{OCT}(v_i, p_k)}{P} \tag{2.4}$$

$\mathrm{rank}_{\mathrm{OCT}}$ produces better selection orders in 99,5% of random graphs compared with rank_u from the *HEFT* approach and reduces the amount of computations [10]. A processor is selected that minimizes $O_{\mathrm{EFT}}(v_i, p_j) = \mathrm{EFT}(v_i, p_j) + \mathrm{OCT}(v_i, p_j)$ *i.e.*, not only the earliest finish time is considered, but also the longest path to an exit vertex.

2.1.4 Bin Packing

In some application use cases, developers expect partitions that simply provide a balanced runnable distribution without the need of dependency consideration. This may be the case if implementations do not necessarily depend on latest updated memory (values) or if deadlines are either easily met in any distribution or deadlines are not required at all. An interesting approach could be using a bin packing algorithm that considers stackability conflicts, *e.g.*, [13]. Instead of *Hanoi* conflicts, that define a single value (*e.g.*, topological order, naming derived in [13] from the mathematical game called *Tower of Hanoi*), directed conflicts further provide specific item sizes (*i.e.*, runnable instructions in our case) to be considered in addition to their topology. The formulas on page 38 in [13] could be used in combination with an *ILP* solver to get optimal solutions.

2.2 New Miscellaneous Features

2.2.1 Hyperperiods

The hyper period is the Least Common Multiple (LCM) of the task periods *i.e.*, the smallest natural number that is an integer multiple of every task period [44], also called major cycle. Considering data transfer rate per hyper-period allows to compare communication cost between various runnables irrespective of their periods [24].

Hyper-periods can be used in App4mc as an alternative to simply grouping runnables referencing the same activation. If these activation groups are not configured, *CPP* and *ESSP* partitioning mechanisms treat runnables without considering their activation reference. Having

a set of runnables with different instruction and activation parameters, their instructions can be matched regarding the overall hyper period in order to form partitions that call runnables multiple times corresponding to their activation references. A simple example illustrates the idea more clearly in Figure 2.1.

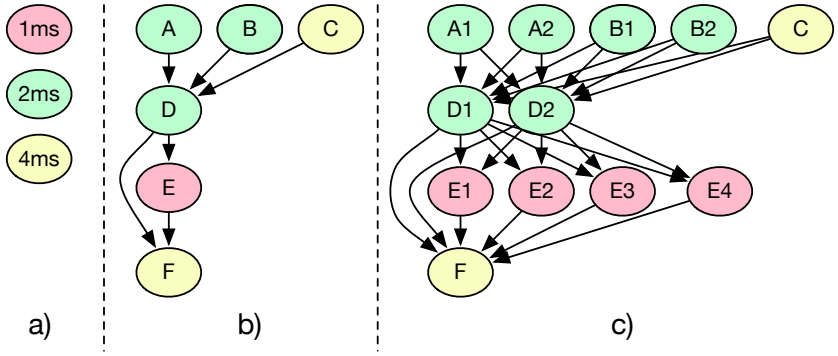

Figure 2.1: Hyper period consideration concept for partitioning

The fictional example (Figure 2.1, *b)*) shows a graph of 6 runnables (A-F) referencing 3 different activations (Figure 2.1, *a)*: A,B,D: 2ms, C,F: 4ms, E: 1ms). The overall hyper period is consequently 4ms. For the purpose of assessing instruction consumption and communication costs while considering different periodic activations, runnables can be instantiated multiple times by the factor their activation period must be multiplied with to match the hyper period value. For example runnable A is instantiated twice since its periodic activation of 2ms must be multiplied with 2 to fit the hyper period of 4ms. Correspondingly, E should be instantiated 4 times since $1ms \cdot 4 = 4ms$. This instantiation result is shown on the right side in Figure 2.1 *c)*. Many additional edges are created that force the partitioning to order the runnable instances sequentially (shown in Figure 2.2 *a)*).

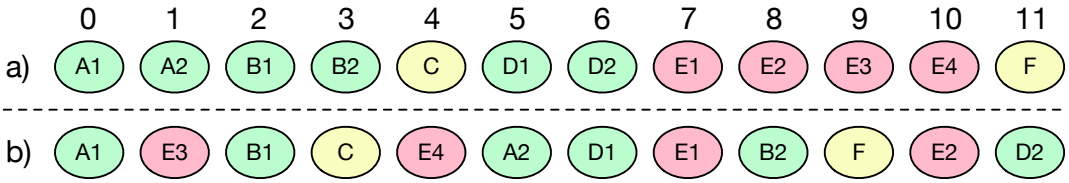

Figure 2.2: Possible hyper period partitioning results of Figure 2.1

However, this may not be desired in some cases and the instances should be stretched along the partition as much as possible. An idea in this regard is to partition the initial graph without the additional instances and insert the instances evenly along the partition afterwards as shown in Figure 2.2 *b)*.

2.2.2 Communication Consideration

In previous releases, communication overheads were assumed equally and their weight was not considered at all. With the new approach in App4mc, not only the amount of intertask communication, but also their cumulated weight is minimized. This new feature becomes most obvious when having a look at Figure 2.3.

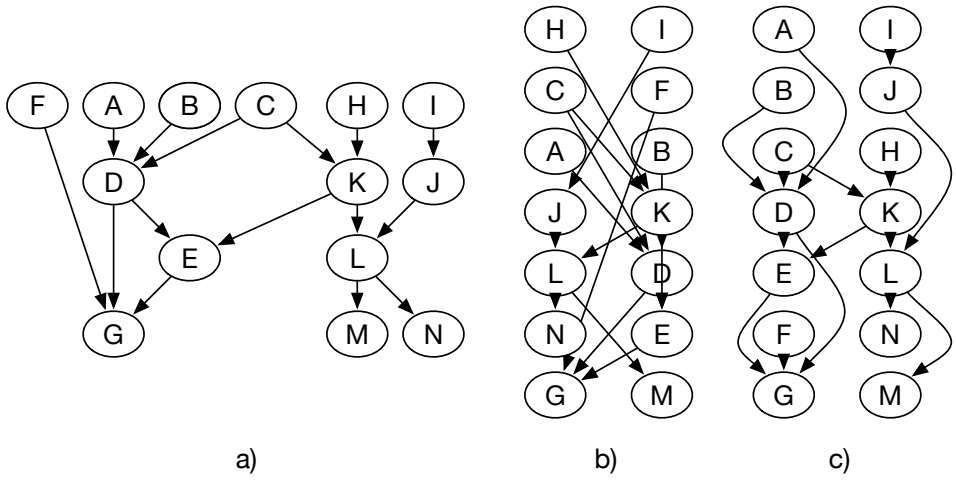

Figure 2.3: a): input graph b): possible result, lots of inter partition communication; c): better result, less inter partition communication, same SLR

The recent approach created solution b) that provides the minimal SLR but has lots of communications between the left side (partition1) and the right side (partition2). The new advanced approach first calculates the best mean SLR via

$$mSLR = \frac{\sum I_R}{\#P}$$

i.e., the sum of all runnable's instructions divided by the number of partitions. Afterwards, the critical path is assigned to the first partition and filled with directly dependent runnables until the partition's sum of instructions equals $mSLR$ or is as near to this value as possible. This assignment calculates metric for the directly dependent runnables to effectively start with runnables that have most communication savings and that have lowest shifting flexibility. To address both of these optimization goals, the shifting value is defined by an inverse percentage parameter that is multiplied with the communication weight and amount. The shift parameter is calculated via

$$sf = 1 - \frac{lit - eit + 1}{mSLR}$$

and the priority via: $priority = (CO_{current} + 2 \cdot CO_{other}) \cdot sf$.

The following fictional example illustrates this optimization: $mSLR = 10$, $cpl = 6$ (*i.e.*, critical path length); 5 directly dependent runnables (each runnable consumes 1 instruction in this example). Consequently just 4 from the 5 directly dependent runnables should be assigned to the initial partition that already contains the critical path. The metrics for the 5 runnables are shown in Table 2.1.

Consequently, the highest rank, i.e., the runnable with ID II, will not be selected for the initial partition. Runnables I,III,IV, and V are inserted right after their predecessor.

Assuming the graph from Figure 2.3, the critical path could be A, D, E, and G. Directly dependent runnables would be F, B, C, and K. With $mSLR = 7$, just three out of F, B, C, and K should be selected for the initial partition and the following metrics of Table 2.2 are calculated.

runnableID	$CO_{current}$	CO_{other}	sf	priority	rank
I	2	2	0,6	3,6	4
II	1	3	0,8	5,6	5
III	2	1	0,8	3,2	2
IV	0	1	0,9	1,8	1
V	1	2	0,7	3,5	3

Table 2.1: Fictional example - metrics for partition assignment to reduce inter partition communication

runnableID	$CO_{current}$	CO_{other}	sf	priority	rank
F	0	1	0,57	1,14	1
B	0	1	0,86	1,71	2
C	1	1	0,86	2,57	3
K	2	2	0,86	5,14	4

Table 2.2: Metrics for partition assignment to reduce inter partition communication for Figure 2.3

Consequently, the algorithm skips K for the initial partition and creates the result shown in Figure 2.3 c). Obviously, the amount of inter partition communication is reduced from 10 to 2 and the algorithm also considers the communication weights.

2.2.3 Backpointer Usage

The usage of back-pointers eases accessing Amalthea models and improves model handling significantly. Especially when dealing with containments, back-pointers can be used to access upward source elements that had to be manually found without back-pointers, *e.g.*, via iterating among all possible source elements. Assuming that a user wants to, *e.g.*, get accessing runnables to a given label, the use of the label's back-pointer references all runnables that access this label. Hence, read-only back-pointers provide a more efficient way to identify model element's containers. Another example can be to get a `ProcessPrototype` that contains a given runnable. This usage saves a lot of processing time and reduces the time until, *e.g.*, the partitioning finishes its execution. This is meaningful especially when using industrial models with huge amounts of model elements.

2.2.4 AffinityConstraint Consideration

The partitioning approaches have been further extended to consider runnable pairings in form of `AffinityConstraints`. These runnable Pairing Entries are supposed to bind runnables together and prevent the partitioning or other processings from separating theses runnables. Such information can be exemplary derived from safety-critical information *e.g.*, functional safety / ASIL as stated in Section 5.2. Furthermore, functional safety information may force runnables to be separated (runnableSeparataionConstraints) due to their resource safety-critical (RSC) properties. In general, affinity constraints can address tasks, runnables or schedulers either to pair or to separate them. Their target can be set to `CallSequences`, cores, tasks or schedulers. Since the partitioning only creates partitions represented by `ProcessPrototypes`, the

affinity constraints' targets are not relevant here. Moreover, only runnable entity groups are considered since neither tasks nor schedulers are usually available or modeled at this design step. The consideration mechanism is realized such that runnable pairings are assumed as a single runnable and their instructions and Label accesses are cumulated. After the partitioning finished its processing, the paired runnables are split back into their original form.

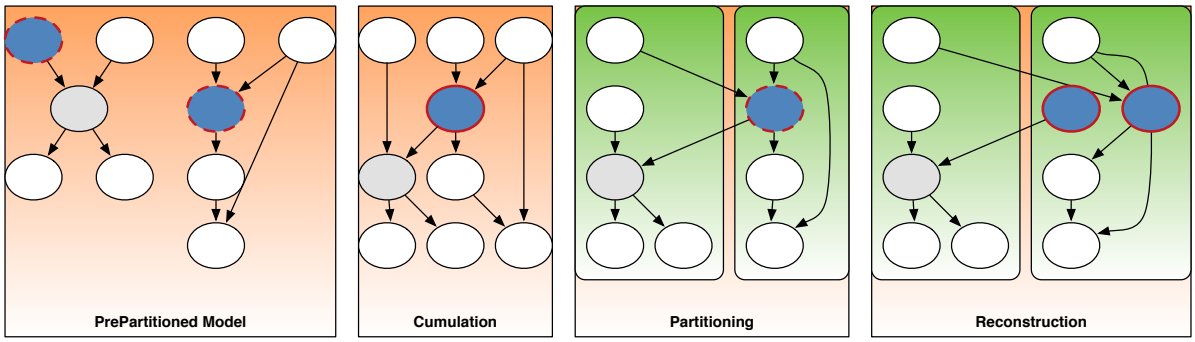

Figure 2.4: AffinityConstraints consideration during partitioning

Figure 2.4 shows the `AffinityConstraint` handling along with an example during the partitioning process from left to right. The most left part shows an pre-partitioned example model consisting of two graphs, various dependencies, and ten runnables in total. The two nodes (*i.e.*, runnables) with dashed (red) stroke are supposed to be paired together in form of an `AffinityConstraint` in order to prevent the partitioning process from separating them into different partitions. For this purpose, each affinity constraint is decomposed into a single runnable with the name 'Cumulatedrunnable' + index that is shown in the 'Cumulation' part (second from the left) in Figure 2.4. Since a runnable pairing may contain mutliple runnables, a cumulated runnable can be composed of multiple runnables. After the cumulation took place, that partitioning approaches are performed normally ('Partitioning' phase in Figure 2.4, second from right). Finally, a reconstruction process needs to decompose the cumulated runnables into the original runnables as seen in Figure 2.4, most right part. The reconstruction needs to

1. delete the cumulated runnables, their runnable calls and RunnableSequencingConstraints containing the cumulated runnables (so all references to the cumulated runnables)
2. add the original runnables from the PrePartitioned model
3. create RunnableSequencingConstraints via all original runnables' label accesses
4. create runnableCalls for all original runnables at the same place of their corresponding cumulated runnable calls

These necessary steps provide the original model structure whilst having runnables that are referenced by runnable pairing constraints in the same partitions correspondingly.

2.3 Workflow

The workflow used in earlier AMALTHEA releases used the MWE2 engine that was replaced by the *Java* EASE script methodology with APP4MC. This further provides MWE2-independent workflow usage while supporting the same workflow structure used up to this point. The partitioning classes accessed by the workflow just required minor changes. An example workflow excerpt is shown in Listing 2.1. A more detailed version can be found at the APP4MC multicore help documentation. Using workflows facilitates the utilization of multiple features in combination with different configuration parameters and provides defining various input files and multiple output files generated by just starting the workflow file as an EASE script.

```
1  bc. Workflow {
2    //setups with 'loadModule(...)' commands; package
       imports; logging configuration; const BASE PROJECT and
       MODEL_LOCATIONs definitions
3    ...

5    var ctx = new DefaultContext()
6    var reader = new ModelReader()
7    reader.addFileName(MODEL_LOCATION1)
8    reader.run(ctx)

10   //prePartitioning (ModelSlot same name)
11   var prepart = new PrePartitioningWrkflw()
12   prepart.setAa(false)
13   prepart.setGgp(false)
14   prepart.setMinimEdge(false)
15   prepart.setEffEdge(false)
16   prepart.run(ctx)
17   //partitioning (ModelSlot same name)
18   var part = new GeneratePartitioning()
19   part.setModelLoc(MODEL_LOCATION1);
20   part.setModelSlot("prePartitioning")
21   part.setPartitioningAlg("essp")
22   part.setNumberOfPartitions("4")
23   part.run(ctx)

25   //Writer
26   var writer = new ModelWriter()
27   writer.setSingleFile(true)
28   writer.setModelSlot("prePartitioning")
29   writer.setFileName("prePartitioning")
30   writer.setOutputDir(PROJECT + "/workflow-output")
31   writer.run(ctx)
32   writer.setModelSlot("partitioning")
33   writer.setFileName("partitioning")
34   writer.setOutputDir(PROJECT + "/workflow-output")
35   writer.run(ctx)

37   print("Finished Workflow")
38   ctx.clear()
39   endWorkflow()
40 }
```

Listing 2.1: Workflow partitioning example

The workflow shown in Listing 2.1 performs the pre-partitioning (line 16) and partitioning (line 23) methodologies whereas different configuration parameters are defined before the run method calls (pre-partitioning lines 12-15, partitioning lines 19-22). The writer component (lines 26-35) finally writes the result models (stored in the model slots) into separate files (lines 31 and 35).

2.4 Visualization

For providing a distinct alternative to *JGraphT* [39] based applet visualization of task graphs or runnable graphs, we investigated different existing frameworks that can be integrated to APP4MC.

2.4.1 Applet

The Applet generation has been extended from the approach presented in [7] to support column wise runnable orders regarding their `ProcessPrototype` reference. *Java* applets can be easily started from web pages to be executed in a *JVM*. However, APP4MC applets have *JRE* as well as *JGraphT* dependency such that they cannot be visualized without access to any of these and further were not directly embedded in Eclipse. Consequently, the following sections outline different possibilities to visualize task graphs or runnable graphs.

2.4.2 Sirius

Sirius is a great modeling framework that provides different easy to use interfaces that can be used for visualization in App4mc. It can be used in combination with any arbitrary EMF models. For this purpose, a viewpoint specification project must be created, that refers the EMF model *i.e.*, Amalthea in App4mc in its Plug-in dependencies respectively its `MANIFEST.MF` file [9]. The viewpoint description file has a '*.odesign*' ending and can have a similar structure to the description shown in Figure 2.5 at the bottom.

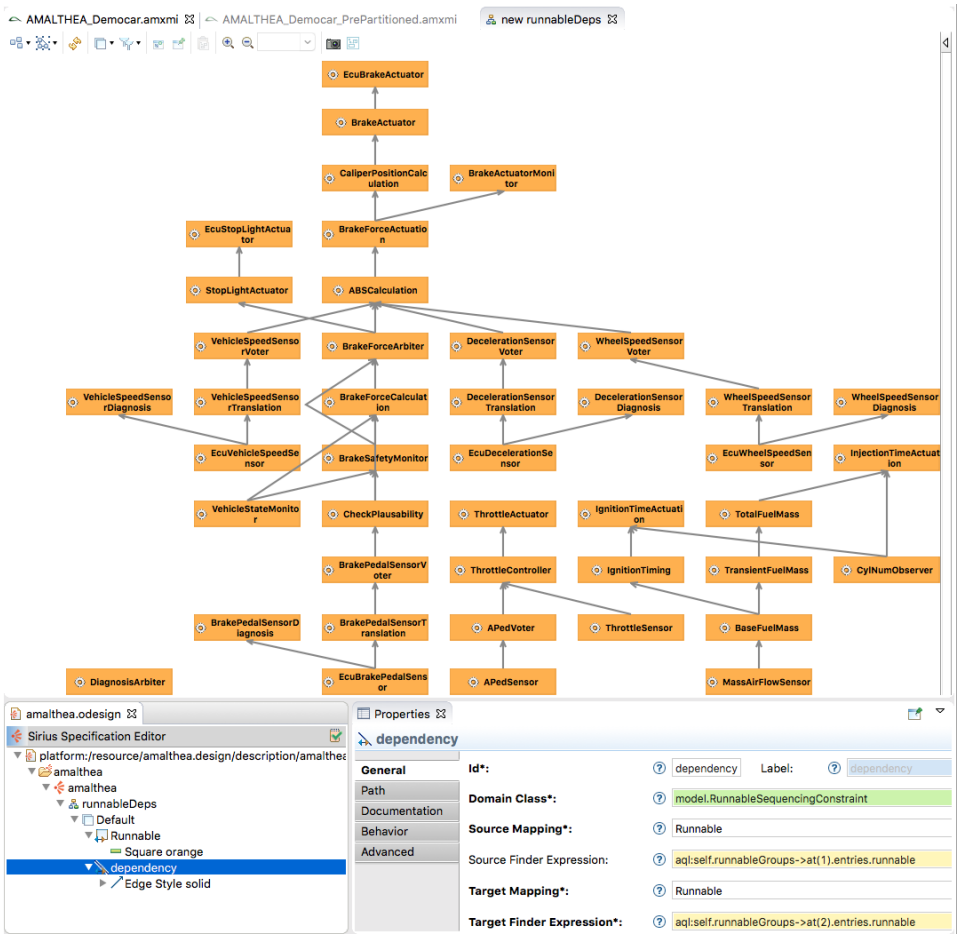

Figure 2.5: Sirius runnable dependency graph

Figure 2.5 illustrates the Democar [26] runnable dependency graph. The dependencies are specified and accessed via the `RunnableSequencingConstraints` as shown in the '*.odesign*' properties for edges in the lower right part of Figure 2.5. The dependency graph is shown in the upper part of the figure whereas the dependencies are shown in form of edges from the runnable source (bottom) to the runnable target (top). Various specifications can be used in order to adjust the visualization. Sirius also provides table based, tree based or sequence based visualizations. Furthermore, the editors can be used to add elements to the model and adjust model entities and properties.

2.4.3 PlantUML

PlantUML is a tool that supports UML based visualization. It requires *Graphviz* [30] and supports online diagram updates based on *Java* or expressions described in [42].

Even EMF models can be interpreted. As soon as the framework has been installed via its update site, the *PlantUML* viewer can be opened and used to visualize the currently selected file from the workspace. *Java* classes are automatically interpreted whereas comment keywords provide further visual possibilities. Assuming the code from listing 2.2, the diagram from Figure 2.6 is automatically generated. Such annotations could be added to generated and partitioned results to allow corresponding *plantUML* visualizations.

```
1  @startuml
2  package ProcessPrototype1{
3    ABSCalculation<<(R, orchid)>>
4    APedSensor<<(R, orchid)>>
5    APedVoter<<R, orchid>>
6    ABSCalculation --> APedSensor
7    APedSensor --> APedVoter
8  }
9  package ProcessPrototype2{
10   class MassAirFlowSensor<<R, orchid>>{
11   write MassAirFlow;
12   }
13   class BaseFuelMass<<R, orchid>>{
14     read MassAirFlow;
15   }
16   MassAirFlow<<L,#FF7700>>
17   MassAirFlowSensor-->MassAirFlow
18   MassAirFlow-->BaseFuelMass
19 }
20 package ProcessPrototype3{
21   IgnitionTiming<<R, orchid>>
22   IgnitionTimeActuation<<R, orchid>>
23   IgnitionTiming --> IgnitionTimeActuation
24 }
25 @enduml
```

Listing 2.2: PlantUML code for visualization

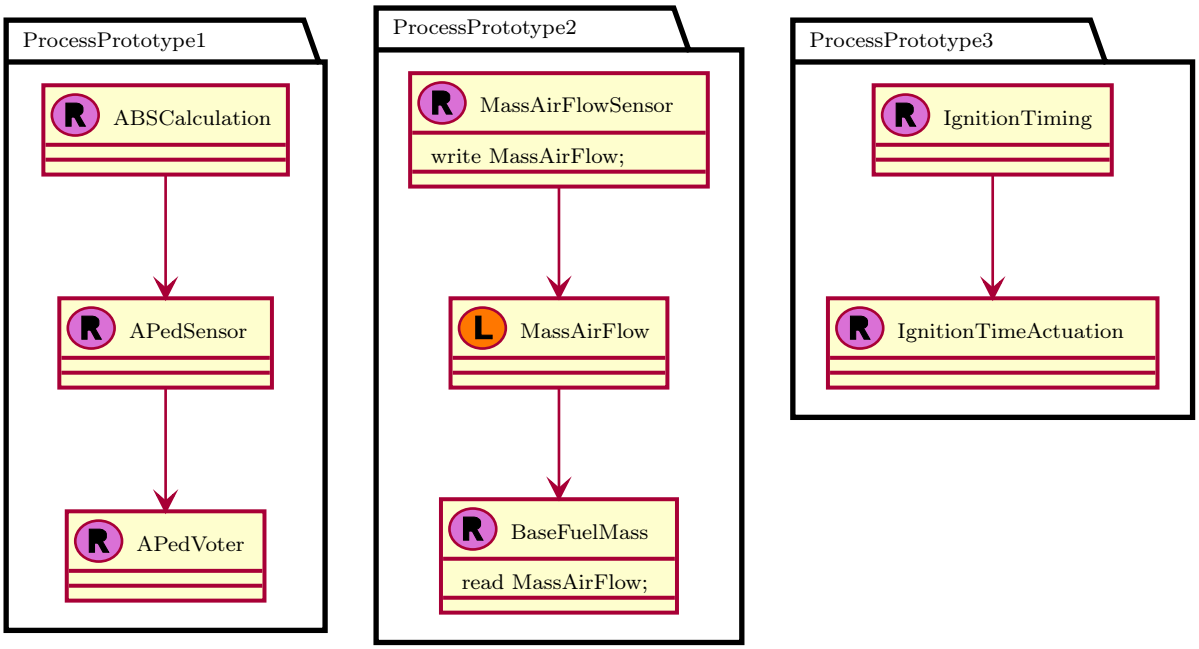

Figure 2.6: PlantUML diagram excerpt: 3 ProcessPrototypes, 7 runnables and one label

Listing 2.2 generates Figure 2.6 that shows 3 `ProcessPrototypes` ('packages'), that vertically show some runnables (from the democar project [26]) and their dependencies are denoted as arrows. `ProcessPrototye2` further illustrates a label between the runnables `MassAirFlowSensor` and `BaseFuelMass`, that is read by the latter runnable and written by `MassAirFlowSensor`.

2.4.4 JGraphT DOT Exporter

Finally, the used *JGraphT* library features a *dot* exporter, that generates a `.dot` file that can be visualized by the *GraphViz* software [30]. Having a graph representation within the *JGraphT* library, it is possible to use the `DOTExporter` from the `jgrapht.ext` library to export a dot file as shown in the following code:

```
1  final VertexNameProvider<Runnable> vnp = new VertexNameProvider<runnable>() {
2    @Override
3    public String getVertexName(final Runnable arg0) {
4      return arg0.getName();
5    }
6  };
7  final DOTExporter<Runnable, RunnableSequencingConstraint> dote = new DOTExporter<Runnable,
    RunnableSequencingConstraint>(vnp, null, null);
8    final Writer w = new FileWriter("dotexport.dot");
9    dote.export(w, cp.getGraph());
10   w.close();
11 }
```

Listing 2.3: Java code accessing the JGraphT DOTExporter class

The source code from listing 2.3 first creates a `VertexNameProvider` (lines 1-6), that is required to express runnable names only for the ellipse style representations in the generated graph. Afterwards, the `DOTExporter` class is instantiated whereas vertices represent runnables and edges represent `RunnableSequencingConstraints`. `.dot` files can be opened in the *GraphViz* software [30] and creates the output illustrated in Figure 2.7.

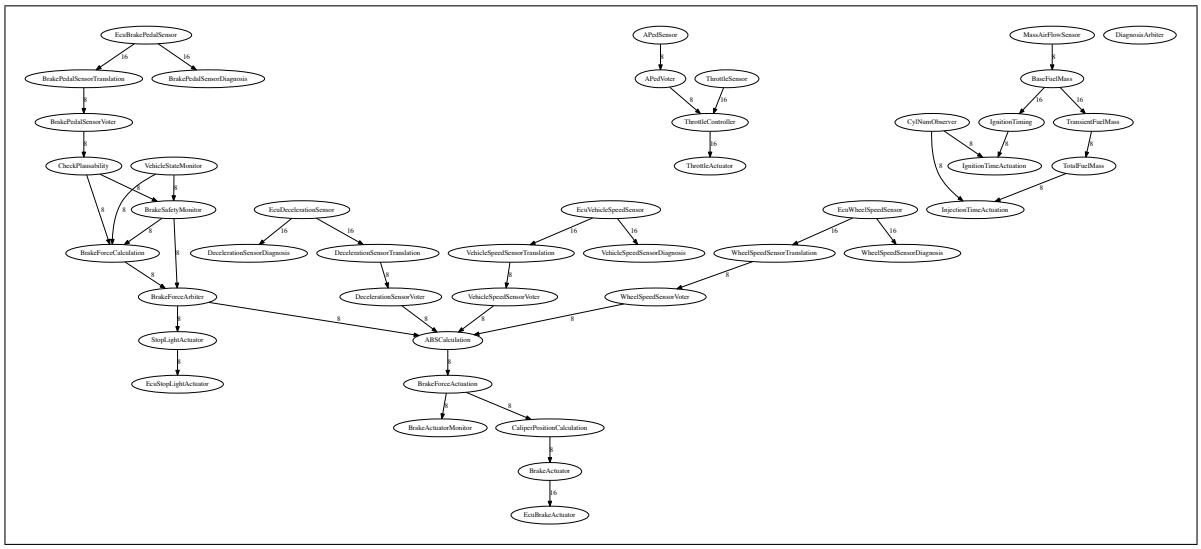

Figure 2.7: GraphViz JGraphT Dot export

Another function (not shown in Listing 2.3) has been implemented, that represents edge names *i.e.*, the number of bits (size) of the label that is accessed by the two runnables (written by the source runnable [`RunnableSequencingConstraints` group 0] and read be the target runnable [`RunnableSequencingConstraint` group 1])

2.5 Summary

The previous sections outline new partitioning strategies, new features such as communication consideration, hyper-periods, `AffinityConstraints`, and workflow accessibility and present

four different runnable dependency graph visualization concepts.

The new partitioning strategies provide the generation of heterogeneous partitions and thereby allow more configuration possibilities in order to generate more effective partitions regarding subsequent mapping to heterogeneous hardware.

The extended and added features described in section 2.2 support advanced runnable binding possibilities, hyper period and communication cost consideration as well as more efficient implementation using backpointers.

In terms of visualization, Sirius and *PlantUML* can easily be embedded to the Eclipse platform. In contrast, the *JGraphT* DOTExporter just requires very few lines for generating the visualizations but requires the external *GraphViz* software for visualization. There has been an Eclipse view plug-in for `.dot` file visualization (`https://github.com/abstratt/eclipsegraphviz`), but it is not maintained and does not work with the newest Eclipse release. *PlantUML* is rather easy to use compared with Sirius, but has limited and less customization possibilities. However, all four possibilities provide runnable dependency graph visualizations that can be used in App4mc. The selection of one of them depends on the desired ease of use and customizability.

3 Mapping

The development of embedded multi-core systems is a complex process with several challenging tasks. One of these tasks is finding a valid allocation of software to hardware, we call this step *mapping*, that combines challenges from various domains. From an embedded point of view, the software has to be executed on very limited resources while meeting (hard) deadlines. Moreover, sporadic events (e.g. interrupts) also have to be considered. From a multi-core point of view, it is necessary to distribute the software to multiple processors or processor cores in due consideration of software dependencies, e.g. task ordering, or specific software requirements, like the availability of floating point units. Moreover, passing information between tasks, which are allocated to different cores, requires valid data paths and communication performance, which depends on the architecture and speed of the communication hardware. This communication can occur at various transfer speeds that depend on the underlying network. The communication network itself can be understood as shared resource, which may delay concurrent accesses, leading to additional bottlenecks if not considered appropriately. Heterogeneous architectures, which are common for embedded systems, increase the complexity of this step even further, since allocations to a non optimal core may change the run-time of the software drastically [37].

In order to cope with this complexity, it is necessary to utilize proper tooling. A tool which aims at distributing software to hardware has been developed within the Amalthea project as *OpenMapping-Plugin* and was already described in deliverable D3.4 [6]. The extensions of this tool with regard to Amalthea4public have been briefly introduced within deliverable D2.1 [8].

This chapter describes the mapping feature within Amalthea4public with a focus on its extension as well as the plugin implementing the mapping functionality as part of the platform App4mc. Moreover, it provides an outlook to the concepts which will be the basis for implementations subject to deliverable D2.3.

Section 3.1 discusses the related optimization methods such as Integer Linear Programing or Genetic Algorithms. The concept of the mapping approach is shown in Section 3.2 along with an evaluation of how different optimization goals (Section 3.2.1), degrees of freedom (Section 3.2.2) as well as constraint handling (Section 3.2.3) can be supported by App4mc. A recommendation on hardware model extensions as part of Task 2.1 is presented in Section 3.3, followed by a description of the adaptations and enhancements that have been performed on the mapping plugin in Section 3.4. Finally, the integrated mapping strategies are outlined in Section 3.5. A summery, which reflects the mapping process, closes this chapter in Section 3.6.

3.1 Related Optimization Methods

Just determining a *valid* mapping of software to hardware often is not sufficient on its own. For instance, if a sequential program is executed solely on the first core of a very performant multi-core system, it may be valid in terms of meeting constraints (deadlines, avoiding deadlocks etc), but it will most likely not utilize any of the multi-core platform's benefits (e.g. concurrency). Consequently, it is desirable to optimize the distribution of software to hardware and eventually

achieve the best result towards one or more objectives, e.g. achieving the best performance (minimizing the execution time), or the most energy-saving distribution (minimizing the energy consumption).

From a mathematical point of view, optimization problems are represented in terms of equations and an objective. The objective describes the goal of an optimization, i.e. which value or function should be minimized or maximized. This can be, e.g., the amount of utilized memory, the overhead of communications, or any other attribute that can be represented by formulas. Further equations are used to define constraints, which allow limiting the solution space. In the context of embedded multi-core systems, such constraints are used for describing the limited resources of the hardware, e.g. the maximal amount of available memory, the limit of concurrently processed tasks, an available bandwidth etc., and ensure the correct allocation of the software to hardware, e.g., by ordering tasks w.r.t. their predecessors, their maintaining deadlines, etc.

Mathematical models can be classified into several optimization problem categories based on their structure. The most common categories as well as optimization techniques that are used in approaches for software to hardware distribution [3] are defined as follows:

- **Linear Programming (LP)** problems are one of the simplest forms of optimization problems and known to contain efficient solving strategies for many problem cases [46]. The general form of LP problems is specified as

  $z = \sum_{j=1}^{n} c_j x_j$

  for an objective z, with n number of variables x with their respective constant c, and

  $\sum_{j=1}^{n} a_{ij} x_{ij} \ \leq \geq = b_i \ \forall \ i \epsilon \{1, \ldots, m\}$

  for constraints, with a, b being constants as well as the comparators '$\leq$', '$\geq$' and '=' for all m number of constraints [46]. As the name of this optimization category suggests, all terms in LP problems' equations must be formed of linear expressions and utilize only real numbers for variables.

- **Integer Linear Programming (ILP)** can be understood as special case of LP problems. In terms of the problem's model, the only difference lies in the usage of integer typed variables in the constraint specifications, which usually make the problem NP-hard to solve. However, commercial and open source solvers are capable of solving many small and medium sized models efficiently [21].

  Due to the nature of many mapping and scheduling problems (allocating a task to an integer number of cores), ILP is a popular approach among those of mathematical optimization for modeling these problems.

- **Mixed Integer Linear Programming (MIP or MILP)** and **Binary Linear Programming (BP)** are further specializations of the ILP problem. The difference in comparison to ILP problems lies in the types of values the variables of the problem may attain, i.e. a mixture of real and integer values (MIP, MILP), or binary (0–1) values (BP). [18]

- **Quadratic Programming (QP)** problems belong to the class of non linear problems and are similar in their structure to LP problems. The objective function consists of quadratic terms instead of linear terms, whereas the constraints must still be linear. The

complexity in solving QP problems is influenced by the number of constraints and the characteristics of the objective function [40], which will either be NP-hard or efficient to solve.

- **Special Ordered Sets (SOS)** can be understood as specific case of an MILP problem. The main benefit in using SOSs lies in a noticeable reduction of the effort, which has to be spend in solving (optimizing) a problem.

- **Genetic Algorithms (GA)** and **Evolutionary Algorithms (EA)** are numerical optimization algorithms inspired by natural selection as well as natural genetics [19]. They are known to be robust and are hence one of the widely used approaches in the software engineering domain [3]. Usually GA begin with a random or guessed *population* of *individuals*, which is encoded as a binary or real-valued string, although *hybrid* approaches also implement other techniques such as *Simulated Annealing* in order to create the initial population.

  Analogue to natural genetics, the population iterates over the operations *selection*, *crossover* and *mutation* (cf. Fig. 3.1). As in natural selection, the *selection* operator promotes fitter individuals more often compared to bad performing. Accordingly, these exchange information is used during the *crossover* operation, e.g. by applying the single point crossover method, which choses a pair of individuals and flips the left half of the information string between those two. Finally, the *mutation* operator flips a single bit of the information string, which helps escaping local optima. Once these operators have been performed, a new population is generated, which provides the basis for the next iteration of the algorithm.

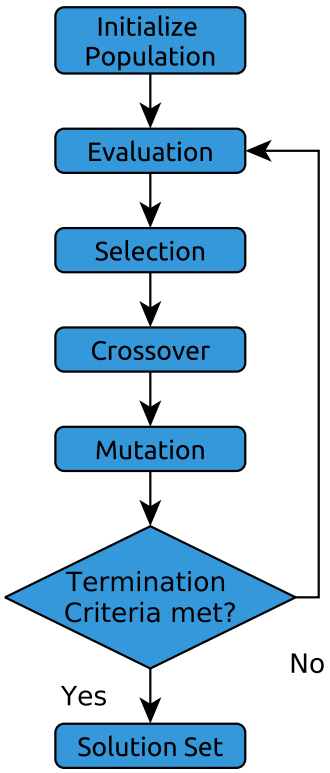

Figure 3.1: GA flow chart

3.2 Concept for Mapping Approach

As part of the extensions within Amalthea4public, the original automated mapping approach [6] has been enhanced by libraries and abstractions for the utilization of genetic algorithm based strategies [37] as shown in Fig. 3.2. Three mandatory input models *(Software, Hardware, Constraints)* provide the required information about the software, hardware, and task ordering for the mapping generation process. An optional model containing constraints *(Property Constraints)* will be used to reduce the solution space of the mapping.

Currently, four mapping strategies have been integrated that can be split into the categories *Heuristic methods*, *Integer Linear Programming (ILP) based methods*, and *Genetic Algorithm based methods*. Unlike ILP based methods, Genetic Algorithm and Heuristic methods, such as the *Heuristic Data Flow Graph (DFG) load balancing*, will immediately create a mapping,

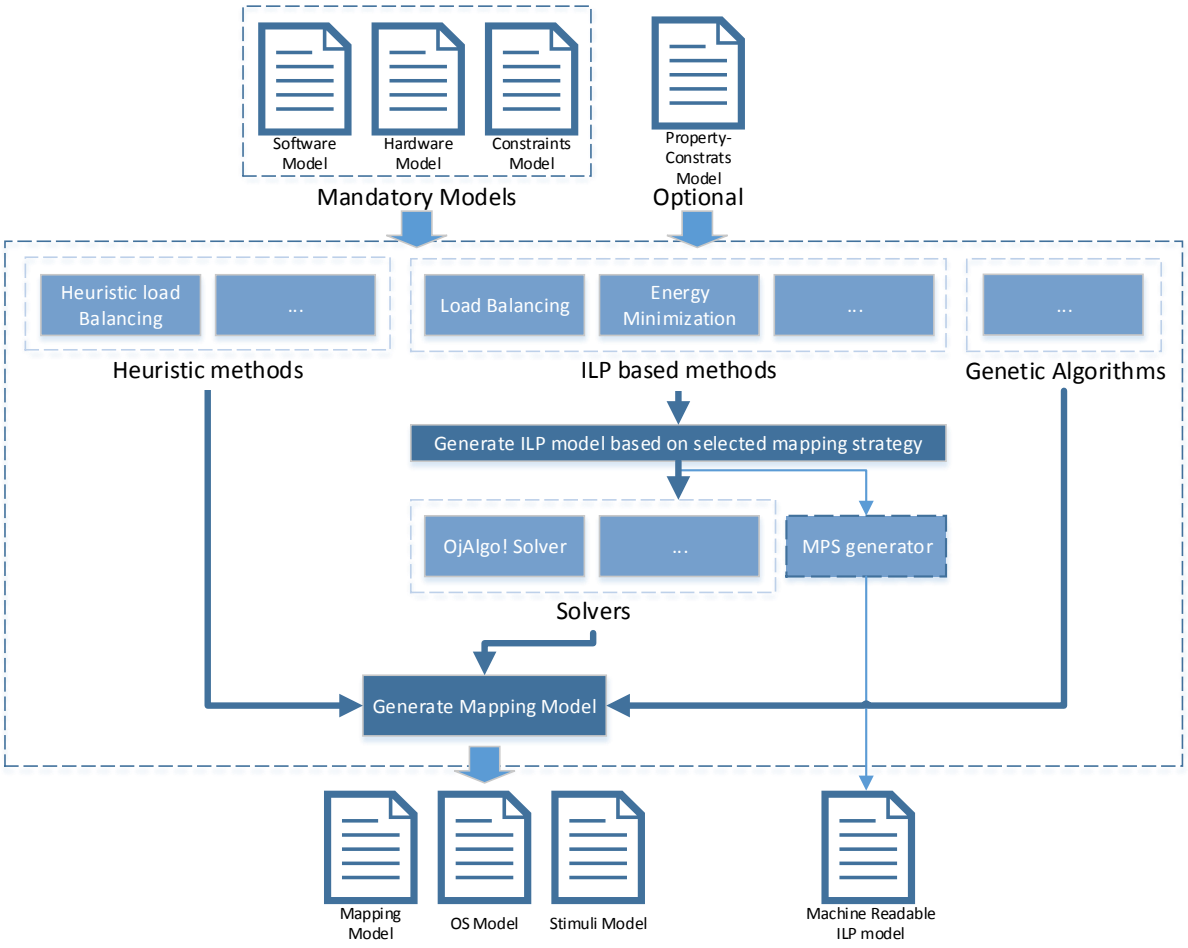

Figure 3.2: Concept for an automated mapping approach using the AMALTHEA Tool Platform

whereas the ILP based methods generate an ILP model of the mapping problem according to the selected mapping strategy. Once the ILP model has been created, it can be solved by one of the mathematical *Solvers*, which is provided by the open source project Oj!Algo [41] in our case. In order to optimize genetic algorithm based strategies, the open source library Jenetics [50] has been integrated.

Contrarily, ILP based methods need to generate an ILP model of the mapping problem according to the selected mapping strategy, e.g. *ILP based load balancing* or *Energy aware mapping* in advance. An optional MPS[1] generator can be utilized for MPS file generation, which contains the whole mapping problem in terms of equations. This allows using external (e.g. commercial) solvers, which tend to be more efficient in solving larger models compared to open source Java implementations, without integrating them into the Tool Platform.

3.2.1 Optimization Goals

As stated in Section 3.1, it is often not sufficient to just find *any* allocation of software elements to hardware components. Instead, solutions which improve one or more aspects of the system,

[1]File format used to store linear programing or mixed integer programing problems

e.g. the energy consumption, reliability, safety or response time, should be found. This section outlines common optimization goals, also referred to as *quality attributes* [3], and evaluates how these goals can be supported by APP4MC and integrated into the platform.

Currently, the approaches integrated into APP4MC support either optimizing the *performance* or the *energy consumption* of the mapping generation's result, which are only a small fraction of the well established quality attributes illustrated in [3] (cf. Table 3.1). Especially in the domain of automotive systems, optimizing the cost, reliability, weight, safety, and security can be crucial.

Table 3.1 shows a quantitative summary of the quality attributes encountered during the literature review of software architecture optimization methods in [3].
The most widespread optimization goals among those methods are the performance, cost and reliability (44% – 37%) of a system. In order to optimize the mapping towards these attributes, it is important to identify how the quality towards these goals can be quantified. The probably most simple approach for this lies in directly annotating these values to an element if they behave statically, e.g. as in cost or reliability of a hardware, which usually remains unchanged by other aspects like software or its distribution. The performance on the other hand usually highly depends on a core's speed, instruction set, and features as well as the nature of the software and its instructions. Hence, it is necessary to provide opportunities to annotate this information as illustrated in Sections 3.5.1 and 3.5.2

Performance	44%
Cost	39%
Reliability	37%
Availability	13%
General	12%
Energy	9%
Weight	3%
Safety	2%
Reputation	2%
Modifiability	2%
Area	2%
Security	<1%

Table 3.1: Summary of Quality Attributes [3]

Less common optimization goals, such as general attributes, weight, safety, reputation, area, and security are usually achieved very similar to maximizing the reliability of a system. In APP4MC, it is possible to annotate each object of the model with so called custom attributes that can obtain any value, which may be evaluated during the mapping process.

A limitation on the model can be observed if more complex information is required, e.g. for minimizing the energy consumption of a system. On an abstract level, an average power consumption may be sufficient for, e.g., identifying an allocation target. However, if the energy minimization should be achieved by configuring voltage levels, it becomes necessary to link multiple attributes with each other, e.g. the consumed voltage level with a certain processing speed on a core. While it is possible to achieve this by e.g. naming conventions, such solutions are prone to errors and should ultimately be implemented by suitable constructs, e.g. fields or structures.

3.2.2 Degrees of Freedom

Optimizing the quality attributes of an embedded system can be achieved by modifying a huge variety of parameters. For example, the runtime of software can highly depend on the implementation, the underlying hardware's speed, the allocation of tasks to cores, their ordering, the mapping and scheduling of communications to networks and data to memories, the accesses to shared resources like peripherals, etc. All those aspects, which are often changed during the optimization process, are usually referred to as *degrees of freedom* (DoF) or *design decisions* in

literature [3]. These can be grouped into several categories depending on the nature's decision, e.g. selecting an element or reordering its nested objects. In order to support these categories within App4mc, several approaches can be utilized as described in the following:

- **Allocation** refers to the deployment of software elements to processing hardware components, although it can be extended to other domains such as mapping data to memories or communications to data paths. Naturally, this DoF can have a significant impact on a wide area of quality attributes depending on the used abstraction level, e.g. the execution speed of a system (heterogeneous cores or boards, mutual exclusion), its reliability (reliability of an ECU), or its cost (different prices of hardware).

  Allocations of executable software to processing cores as well as mappings of data to memories are already covered by Amalthea's mapping model. Use cases for distributing communications to channels as well as higher abstractions of allocations (e.g. ECUs), however, are not yet considered and consequently require an extension of the models.

- A **scheduling** can be described as the mapping of operations, such as executing tasks, passed information between tasks, or resource accesses, to a specific time interval on a processing unit. This usually allows, e.g., preventing mutual exclusions, ultimately leading to better performance, lower cost, or reduced energy consumption.

  Amalthea distinguishes between the bare ordering (e.g. mapping to timeslots) and the previously mentioned allocation to a target. While the latter is already covered by the mapping model, the ordering is mainly influenced by sequencing/ordering constraints and activation rates. Basically, each operation that can be expressed in terms of a runnable or read/write operation can also be reordered.

- Modifying **parameters** allows to change the individual properties of software as well as hardware. For instance, a core may provide unique voltage levels and can be executed at a slower frequency while consuming less energy compared to its default value. These constructs can be supported by custom attributes in almost all classes of the Amalthea models, which allows annotating additional information to an object.

- **Clustering**, or the agglomeration of tasks, describes the process of grouping tasks in order to reduce the complexity of allocations. From a technical point of view, the agglomeration process will likely consist of merging tasks by extracting `ProcessPrototype`'s runnable calls and forming new `Task`s with a different runnable call distribution. Since the task creation method within App4mc already performs such an operation, except for modifying the distribution from runnables to tasks, this DoF can be fully supported and represented by the Amalthea models.

- HW/SW **partitioning** describes the distribution of an application's computations on a microprocessor (software) and IC fabrics (hardware). It is mainly used to increase the performance of a system and/or to reduce its cost.

  While it is possible in Amalthea to allocate tasks to specific cores, it is not possible to distinguish which part of software should be implemented as software and which as hardware. In the current model version, an alternative may be to specify a `ComplexNode` acting as, e.g., a FPAG or an ASIC as allocation target, although it is desirable and planed to extend the hardware model to support this use case natively.

- The **selection** of an element during the mapping process can be interpreted in various ways, e.g. selecting a software module, a suitable hardware, or the best suited implementation (cf. HW/SW partitioning). Since the latter has already been covered within this chapter, we reduce the scope of the DoF selection to selecting a best suited element, either software or hardware, out of various possibilities.

  Although the selection of software or hardware is not yet considered by the plugins within Amalthea, additional plugins, such as the variability tool, aim at determining a compatible hardware in an early design phase, thus, it can be considered as hardware selection on a higher abstraction level. The models of Amalthea currently do neither provide constructs for selecting hardware nor selecting software except for the mapping model. However, the mapping process may *re-configure* a software or hardware model to support such modifications.

- The category of **replication**s refers to operations regarding changing the multiplicity architectural elements in either hardware, software, or both. Examples for this are listed as increasing the redundancy by copying a hardware along with the executed software (hardware replication) or by n-version programing (software replication). While the models within Amalthea naturally allow to create copies of any given element in software and hardware, it is not yet considered by the mapping process itself.

3.2.3 Constraint Handling

When developing embedded systems, the design space is naturally restricted, since not all combinations of the DoFs can be applied or lead to feasible results. For instance, safety critical systems will likely be kept on the same ECU while redundancy will usually be achieved by doing the exact opposite. The limitations introduced by the hardware, e.g., the memory size, or the software, e.g., deadlines, restrict the solution space even further.

The most common constraint handling techniques according to [3] can be categorized into the groups prohibit, penalty and repair:

- **Prohibit** refers to optimization methods that discard a solution if any constraint is violated.

- Another approach in handling constraints lies in converting the constraint optimization problem into an unconstrained, while adding a (numeric) **penalty** to the final fitness value.

- Finally, some approaches apply a **repair** mechanism during the optimization method if constraints are violated. However, this requires additional knowledge about the problem in order to repair an solution, which ultimately leads to an additional overhead to the algorithm.

The mapping process within App4mc currently applies mapping strategies based on either ILP or GA. Both approaches are well known to support constraint handling techniques that either prohibit solutions violating constraints and add penalties on their fitness value, whereas repair mechanisms will only be feasible in GA based strategies.

3.3 Hardware Model Extensions

3.3.1 Declarations and Instances

Amalthea's Hardware Model distinguishes between instances of a hardware element and its type declaration, which has several benefits in describing multi- and manycore systems, e.g. a 128 core processor. Since each instance of these cores may inherit the properties of a common core (proto-)type, the modeling overhead can be significantly reduced. Moreover, the type's properties also allow modifying the properties of all cores at once.

One major downside of the current implementation lies in the distribution of attributes between types and instances. Since some properties can only be specified within an instance *or* its prototype, it is not possible to choose their location freely or even overwrite their inherited values. In order to improve this behavior, we propose the following changes for all hardware element abstractions (Systems, ECUs, Microcontrollers, Cores, Memories, and Networks):

- All attributes will be moved to the class representing an hardware element's prototype.
- An instance will become a specialization of its prototype class.
- If an attribute has been set within a prototype, it is inherited by the referring instance, unless it is being overwritten by a different value in the latter.

```
2  class HWModel extends BaseObject {
3    contains Tag[] tags
4    // containments of all hardware type descriptions, i.e. Core(s), Memory(s), Network(s), ...
5    contains HwSystemPrototype[] systemPrototypes
6    contains ECUPrototype[] ecuPrototypes
7    contains MicrocontrollerPrototype[] mcPrototypes
8    contains CorePrototype[] corePrototypes
9    contains MemoryPrototype[] memoryPrototypes
10   contains NetworkPrototype[] networkPrototypes
11   contains AccessPath[] accessPaths
12   // Containment of the system described by this model
13   contains HwSystem system
14 }

16 // ...

18 class MicrocontrollerPrototype extends ReferableBaseObject, ITaggable
19 {
20 }

22 class Microcontroller extends ComplexNode, MicrocontrollerPrototype
23 {
24   refers MicrocontrollerPrototype[0..1] inheritsSettingsFrom
25   contains Core[+] cores
26 }

28 class CorePrototype extends ReferableBaseObject, ITaggable
29 {
30   int instructionPerCycle
31   int lockstepGroup
32   contains DataUnit bitWidth
33 }

35 class Core extends ComplexNode, CorePrototype
36 {
37   refers CorePrototype[0..1] coreType
38 }
```

Listing 3.1: Extract of refined Hardware Model

An example illustrating these changes on an excerpt of the Hardware Model regarding the microcontroller and core elements is shown in Listing 3.1.

The containments in Lines 5 – 11 represent holders for type descriptions of all hierarchies (System, ECU, Microcontoller, and Core) and elements (Memory and Network). Although a

model will only consist of one System, the `SystemPrototype` will be useful in storing multiple settings for systems as well as switching between these.

Lines 18 and 28 show the type definitions for microcontrollers and cores. These contain all attributes and properties of an element. Compared to the actual instance of their corresponding element (Lines 22 and 35) they lack of any contained elements as well as reference to a type where properties are inherited from. The strict separation of instances and types allows to simplify model handling, since it will prevent overheads due to cycles in inheriting attributes (A inherits from B, which inherits from A) or long chains (A inherits from B, which inherits from C, ...).

3.3.2 Metric and IEC prefixes

Several attributes of the components within the hardware model are annotated with large numeric values, e.g. a quartz's frequency (usually in terms of MHz or GHz) or memory sizes (currently up to TB). Annotating such elements without proper prefixes may lead to error prone and impractical models, e.g. if the developer accidentally misses a single digit of a large number. Moreover, digital information is usually scaled by 1024, which makes raw numeric values difficult to read on greater memory sizes (e.g. 33554432 Bytes instead of 32 MiB).

```
1  enum BinaryPrefix {
2    _undefined_ // Display: "<unit>"
3    BYTE as "Byte" = 0
4    KIBI_BYTE as "KiB" = 1
5    MEBI_BYTE as "MiB" = 2
6    GIBI_BYTE as "GiB" = 3
7    TEBI_BYTE as "TiB" = 4
8    PEBI_BYTE as "PiB" = 5
9    EXBI_BYTE as "EiB" = 6
10 }
```

Listing 3.2: Enumeration list for ByteSize date type

While metric prefixes have already been added to Amalthea's models, IEC prefixes, which are mainly required for representing a memories size, prove to be slightly more challenging in their implementation. In comparison to metric prefixes, infeasible byte sizes are much easier to forge using IEC prefixes due to their scale of 1024, e.g. if 0.3 KiB are annotated (resulting in 307.2 Bytes).

Our proposition for a type, which provides IEC prefixes and the essential operations for getting correct scaled values, is shown in Listings 3.2 and 3.3. The class consists of two attributes (value and the binary prefix) and provides an additional method for automatically converting these into scaled bytes (7 – 11), which are also displayed within the Amalthea model editor (12 – 15).

```
1  class ByteSize {
2    double value
3    BinaryPrefix unit
4    op String toString() {
5      return value + " " + if(unit == BinaryPrefix::_UNDEFINED_) "<unit>" else unit.literal
6    }
7    op BigInteger getByteScaledValue() {
8      if(unit != null && unit != BinaryPrefix::_UNDEFINED_) {
9        return new BigDecimal(value).multiply(new BigDecimal("1024").pow(unit.value)).
           toBigIntegerExact();
10     } else return null
11   }
12   @GenModel(propertyCategory="Read only (Informational Purpose)")
13   derived readonly BigInteger Bytes get {
14     return getByteScaledValue();
15   }
16 }
```

Listing 3.3: Code for ByteSize

```
+--------------------------------------+------------------------------------------------+----------------+------------------------+
| Core              (#Tasks/#Runnables) | Utilization                      (Percentage) | Cycles         | Time (round to 5 fig.) |
+--------------------------------------+------------------------------------------------+----------------+------------------------+
| e200z1?type=Core          (  1/   9) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (100%) | 880.000        |        15.172,41379 µs |
| e200z7?type=Core          (  2/  11) | XXXXXXXXXXXXXXXXXX                      ( 49%) | 880.000        |         7.586,20690 µs |
| e200z0?type=Core          (  2/  11) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (100%) | 880.000        |        15.172,41379 µs |
+--------------------------------------+------------------------------------------------+----------------+------------------------+
```

Figure 3.3: Visualization of mapping results

3.3.3 Addressing

All types of memory addresses (e.g. offsets or absolute addresses) have been annotated as long integer typed values, which usually are incapable of addressing any location beyond 2GB. In order to cope with this, `BigInteger` values have been determined as the default value for these types of attributes.

3.4 Tooling Extensions

This section outlines the adaptations as well as enhancements that have been performed on the mapping plugin in order to cope with the modifications of the App4mc tool platform as well as its Amalthea models.

3.4.1 Library supporting Genetic Algorithm

One of the long-term goals within Amalthea4public was the integration of additional optimization techniques, which can cope with larger problem sizes or heavy constraint problems. Genetic algorithms have already proven to be well suited for such an application. In order to support GA bases strategies, integrating a library for evolutionary algorithms into App4mc was required. Jenetics is a modern Genetic Algorithm and Evolutionary Algorithm library written in Java, which supports Java8 and utilizes its extensions, e.g. Java streams. Providing a clear separation of the concepts of genetic algorithms, among others, fitness functions and chromosomes, Jenetics is highly adoptable and capable for its integration to mapping strategies of App4mc.

3.4.2 GA based Mapping Strategies

A comparatively simple genetic algorithm based load balancing approach has been integrated into App4mc. It utilizes the java library Jenetics and is further described in Section 3.5.4.

3.4.3 Refactoring and Visualization API

As part of the refactoring work on the mapping plugin, multiple APIs have been created to simplify the integration of new optimization approaches as well as mapping strategies. For instance, a package for agglomerating the allocations of tasks to cores into the mapping model has been created, which is additionally capable of creating a textual output describing the mapping results (cf. Fig. 3.3), e.g. the utilization of each core or the number of allocated runnables and tasks. Consequently, this helps to provide a common and comparable output among the mapping strategies.

3.4.4 Java8

Including the Jenetics library as well as the recent release of APP4MC lead to an upgrade of the required Java version from 7 to 8, which brought several benefits and additional features that further allowed improving the mapping plugin. Among others, we could simplify multiple parts of the plugin by utilizing lambda expressions and streams, e.g. for array and search operations. Moreover, parallel streams allowed to parallelize large parts of the plugin, which provides a significant improvement on the computation speed.

3.4.5 Workflow Examples

The workflow support of the mapping plugin allows automating the mapping process by specifying the input and output files as well as the operations (e.g. the mapping strategy) that should be performed on the models. We extended the workflow to cover all mapping strategies and created an additional workflow example for the *energy minimization examples*. It creates two mappings for the HVAC as well as Democar software models on an i.MX6 board, while minimizing their individual power consumption.

3.4.6 Miscellaneous Changes

The following additional modifications, which do not fit in any of the previous categories, have been implemented and are part of the current APP4MC platform.

- OjAlgo Upgrade: The library for solving ILP based strategies has been updated from version 35.5 to its most recent version (39.0), which brings a large number of improvements to the utilized package *org.ojalgo.optimisation*, e.g. bug fixes and Java8 support.[2]
- Operating System generation: Due to a change in AMALTHEA's `OSModel`, an empty Operating System instance is created during the mapping generation process.
- Namespace: The namespace of the mapping plugin has been changed for the eclipse project APP4MC into `org.eclipse.app4mc.multicore`.
- Reconfiguration: A minor group of mapping problems could not be optimized using the OjAlgo solver. For this reason, a pragmatic reconfiguration approach has been integrated, which extends the capability of the Solver on this group of models.

3.5 Implemented Mapping Strategies

This section outlines the already existing mapping strategies that are part of the AMALTHEA4PUBLIC project (Sections 3.5.1 – 3.5.3) and describes the newly integrated genetic algorithm based approach (Section 3.5.4).

3.5.1 LPT Greedy Load Balancing

The *Longest Processing Time Greedy Scheduling* (LPT Greedy) strategy sorts the tasks in descending order of their instruction values and performs a greedy scheduling. While this heuristic does not consider any constraints in the ordering of the tasks, which makes it only

[2] See: https://github.com/optimatika/ojAlgo/wiki/v39

applicable on independent groups of tasks, it has a very low runtime and supports heterogeneous architectures by estimating a task's runtime based on its number of instructions and the core's *Instructions per Second*. The goal of this strategy is to minimize the overall execution time of all tasks by allocating tasks to processing elements (i.e. cores).

The pseudecode for this algorithm is shown in the listing on the right (Algorithm 1) with m denoting the number of tasks, n the number of cores, and $p_1, p_2, ..., p_m$ the list of processing times for each task $1-m$. Initially, all tasks are sorted in descending order (line 1) before all cores are initialized with no load (line 3) and an empty list of allocated tasks is initialized (line 4). The scheduling algorithm begins with the task having the longest execution time (line 6), and allocates it to the core with the lowest load (line 8) which is determined in line 7. Afterwards, task's load is added to the selected core (line 9) before the algorithm repeats the process for the next task.

Data: m, n, p_1, p_2, ..., p_m
Result: Allocation of tasks to cores
1 Sort jobs so that $p_1 \geq p_2 \geq ... \geq p_m$;
2 **for** $i \leftarrow 1$ **to** m **do**
3 $\quad L_i \leftarrow 0$;
4 $\quad J(i) \leftarrow \emptyset$;
5 **end**
6 **for** $j \leftarrow 1$ **to** n **do**
7 $\quad i = \arg\min L$;
8 $\quad J(i) \leftarrow J(i) \cup j$;
9 $\quad L_i \leftarrow L_i + p_j$;
10 **end**
Algorithm 1: LPT Greedy Algorithm

3.5.2 ILP based Load Balancing

The *ILP based Load Balancing* strategy is the integer linear programing (ILP) based equivalent of the load balancing strategy described in Section 3.5.1. This approach creates a simple mathematical model describing the *machine scheduling problem*, which is optimized by the integrated Oj!Algo solver. Like the heuristic load balancing approach, this strategy targets heterogeneous core architectures and minimizes the upper bound of the execution time among all cores by allocating tasks to cores.

3.5.3 ILP based Energy Minimization

ILP based Energy Minimization is the second ILP based strategy. It is based on the work of Zhang et. al. [51] and implemented as a two phased approach, where the first phase executes a heuristic that allocates the executable software to cores while maximizing the *slack*. These slacks are idle time slots on a core, i.e. slots where no software is being executed because a task waits for the results of a predecessor on another core. The ILP part of this strategy is finally used to slow down the cores without harming any deadlines.

Compared to the other approaches, there are several major differences. The goal of this strategy is minimizing the total energy consumption while meeting all deadlines, e.g. the software will not be executed as quick as *possible*, but quick as *necessary*. Moreover, it focuses on the distribution of runnables, i.e. runnables are distributed on the cores of a target platform. While this strategy considers constraints about the predecessors of the runnables, it does not support heterogeneous architectures, e.g. only cores with an equal configuration are supported. The output of this strategy is an allocation of tasks to cores, as well as the number of instructions for each of the selected voltage levels on each core.

3.5.4 GA based Load Balancing

As part of the extensions within AMALTHEA4PUBLIC, a *genetic algorithm based load balancing* strategy has been implemented to the APP4MC platform. Naturally, it allows minimizing the overall execution time of a given application by determining the allocations of $T \rightarrow C$ for each task T to a core C.

The initial population of this algorithm is set to 1000 and initialized with random values. Each chromosome contains one gene for each task describing the core it is allocated to, e.g., the value 0 expresses an allocation to the first core, the value 1 to the second core etc., whereas the gene's index equals the task's index. Due to this encoding, it is not possible to forge technically invalid chromosomes in an unrestricted design space.

The fitness function evaluates the longest execution time of the sum of tasks among all cores, i.e., $max(T_1, T_2, T_3, ...)$ with T_x being the execution time on a core x. Consequently, the algorithm aims at minimizing the numeric value of the fitness function by applying a single point crossover operator as well as mutations. Similar to the random initialization of the population, these operations always produce valid chromosomes due to their encoding.

3.6 Summary

This chapter provides an outlook to the mapping concept for Multi- and Manycore Systems along with a discussion of the major components involved in creating an optimal distribution of software to hardware.

Section 3.1 outlines and compares several optimization methods, which may be utilized to generate optimized distributions of software elements to hardware components. We identify and select genetic algorithm based approaches as the best suited and most expandable candidates for further mapping strategies due to their benefits in speed, robustness and wide area of available strategies, and extended the mapping process accordingly as described in Section 3.2.

Moreover, we evaluate how future strategies can be implemented into APP4MC by analyzing various optimization goals, degrees of freedom, and constraint handling techniques (cf. Sections 3.2.1 – 3.2.3). This allows us to determine the restrictions given by the AMALTHEA models as well as to provide recommendations for future extensions to the hardware model in Section 3.3.

The extensions that have already been realized as part of AMALTHEA4PUBLIC and implemented in APP4MC are described in Section 3.4. This covers the library for handling genetic algorithm based mapping strategies and provides a description of the GA based mapping strategy, followed by the refactoring process, common visualization among all mapping strategies, Java8 and its parallel computation support, additional workflow examples as well as the other minor modifications.

Finally, all supported strategies are outlined in Section 3.5 along with a detailed description of the recently implemented GA based load balancing strategy.

4 Scheduling for ECU Networks

In this chapter, we present an extended development process that enables partition and mapping of runnables to Electronic Control Unit (ECU) networks based on MechatronicUML and App4mc.

In Amalthea4public, we aim to integrate App4mc and MechatronicUML. On the one hand, MechatronicUML provides a sophisticated compositional verification approach allowing to formally verify behavioral models that have been modeled on Platform Independent Model (PIM) level by means of model-checking. However, the deployment of MechatronicUML models on multi- and many-core target platforms has not been considered in detail and MechatronicUML lacks the timing analysis capabilities of App4mc for multi- and manycore systems. On the other hand, Partitioning and Mapping in context of ECU networks needs to be done manually for each ECU right now in Ḃy integrating App4mc and MechatronicUML, we aim to provide an exemplary tool chain from software design over Partitioning and Mapping to deployment on ECU networks and timing analysis.

In Section 4.1, we introduce the most relevant foundations for this chapter, namely an overview of the MechatronicUML development process and the MechatronicUML Allocation Specification Language (ASL). In Section 4.2, we present the extended MechatronicUML development process developed in context of Amalthea4public. Here, we focus on the allocation of software components and their runnables to ECU networks. Finally, in Section 4.3, we present the detailed concepts for the extended partitioning and mapping approach. The presented work is an extension to [2], especially sheding a light on the allocation of software components and runnables.

4.1 Foundations

Figure 4.1 gives a brief overview of the MechatronicUML development process. Here, we focus on the fourth step.

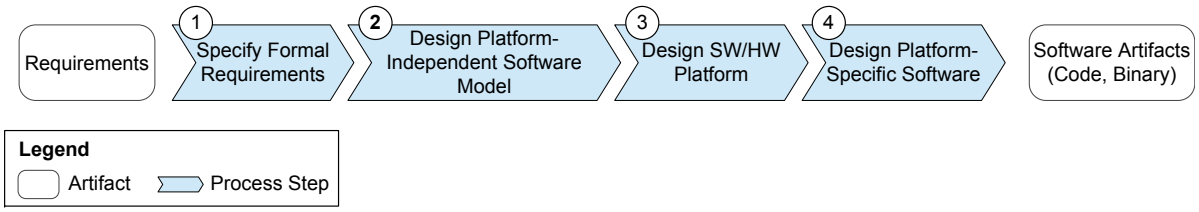

Figure 4.1: MechatronicUML Development Process (based on: [22])

The first step when developing with MechatronicUML is the scenario-driven specification of formal software requirements (cf. [2, 31]).

Based on the formal requirements, the platform-independent software development starts in step 2 by modeling a component-based software architecture using the MechatronicUML domain-specific language.

In step 3, the target platform, described by a Platform Description Model (PDM), is modeled by the platform engineer. A platform describes an execution environment, *i.e.*, the used hardware (resources like ECUs, sensors, and actuators), operating system, and additional software that is needed for the execution of the software system.

Step 4 concludes with the actual design of the platform-specific software. One part of this step we focus on in the following is the allocation of software components from step 2 to the different ECUs.

Please refer to [1, Section 3.4.6] and [22] for more detailed descriptions on the MechatronicUML development process.

The amount of possible allocations of a set of software components to a set of ECUs is exponential regarding the amount of software components. This complexity can be reduced by defining mandatory and useful allocation constraints. MechatronicUML provides a model-driven allocation engineering approach enabling an allocation engineer to specify such allocation constraints in an easy and expressive way. The core part of this approach is the ASL, which enables allocation engineers to define an allocation specification, for existing software architecture and hardware platform models. The ASL provides a library with Object Constraint Language (OCL) operations that ease the creation of the allocation specification. The allocation specification is automatically transformed to an Integer Linear Programming (ILP) that encodes the allocation problem. The resulting ILP is solved by LPSolve. Finally, the solution of the ILP is transformed to an allocation model via a back-transformation that maps software components to ECUs. As a result, the solving of the whole allocation problem becomes transparent for allocation engineers. They get a feasible solution without having to know how to encode and solve the allocation problem as a large and complex ILP.

The ASL provides four commonly used allocation constraint kinds. The constraint kinds are *collocation*, *separateLocation*, *requiredLocation*, and *requiredResource*, which we shortly introduce in the following.

The *collocation* constraint is used to specify that two software components, *e.g.*, named sc11 and sc12, have to be allocated to the same ECU (cf. Listing 4.1). Thereby, it is possible to avoid safety-critical communication between components via an unreliable bus or to reduce communication latencies. The constraint evaluates to a set that consists of a 2-tuple whose concrete type is defined by the descriptors. The elements of the tuple can be accessed via the names firstComponent and secondComponent. We use the OCL operation allocateToSameECU(instance1 : String, instance2 : String) to define the allocation constraint that the components with the name sc11 and sc12 have to be collocated. Later, the evaluation result of this operation call is transformed to corresponding ILP constraints. The OCL operation allocateToSameECU(...) is stored in the MechatronicUML specific allocation operation OCL library.

```
1 constraint collocation collocateSC11AndSC12 {
2   descriptors (firstComponent, secondComponent);
3   ocl   self.allocateToSameECU('sc11','sc12'); }
```

Listing 4.1: A Constraint of the Kind collocation

The *separateLocation* constraint is used to specify that two components, *e.g.*, named sc6a and sc6b, have to be allocated to different ECUs. Thereby, it is possible to avoid that redundant components are allocated to the same ECU and fail at the same time in the case of an ECU hardware fault. Listing 4.2 shows this constraint with the name separateLocationSC6aAndSC6b

of the kind **separateLocation**. The constraint evaluates to a set that consists of a 2-tuple. The elements of the tuple can be accessed via the names firstComponent and secondComponent. We use the OCL operation allocateToDifferentECUs(instance1 : String, instance2 : String) to specify that the components with the names sc6a and sc6b cannot be collocated. The allocateToDifferentECUs 2-tuple is syntactical the same as the allocateToSameECU 2-tuple. Due to the distinguished kind its semantics is the exact opposite.

```
1 constraint separateLocation separateLocationSC6aAndSC6b {
2   descriptors (firstComponent, secondComponent);
3   ocl   self.allocateToDifferentECUs('sc6a','sc6b'); }
```

Listing 4.2: A Constraint of the Kind separateLocation

The *requiredLocation* constraint is used to specify that a component has to be allocated to specific ECUs or ECUs that are part of specific platforms. Thereby, it is possible to avoid that safety-critical components are allocated to non-secure ECUs that are designed for entertainment purposes and have, *e.g.*, no trusted platform with appropriate authentication mechanisms for access control. Additionally, we have to ensure that components, which communicate with each other, are allocated to the same ECU or to ECUs that are connected via a bus or a direct link. Listing 4.3 shows this constraint with the name requiredLocationSC8 of the kind **requiredLocation**. The constraint evaluates to a set that consists of several 2-tuples. The elements of a 2-tuple can be accessed via the names component and allowedResource. We use the OCL operation allocateComponentToPlatform(component : String, platform : String) to specify that the component with the name sc8 has to be allocated to one of the ECUs of the Brake platform.

```
1 constraint requiredLocation requiredLocationSC8 {
2   descriptors (component, allowedResource);
3   ocl   self.allocateComponentToPlatform('sc8', 'Brake'); }
```

Listing 4.3: A Constraint of the Kind requiredLocation

The *requiredResource* constraint is used to specify that an allocation has to respect certain resource restrictions. Thereby, it is possible to avoid that more main memory of an ECU is used by software components than an ECU provides. Thus, we have to specify an OCL expression that returns for each ECU the available memory and a set that describes the memory consumption of each component, if it is allocated to that ECU. Listing 4.4 shows a constraint with the name maxMemoryConsumption of the kind **requiredResource** that is used to guarantee that the components, which are allocated to the same ECU, do not exceed the ECU's available memory. The constraint evaluates to a set that consists of nine 2-tuples (one 2-tuple for each ECU). Such a 2-tuple has the named parts availableMemory and requiredMemory. The named part availableMemory represents an ECU's available memory. The named part requiredMemory refers to a set that consists of 3-tuples. Such a 3-tuple has the named parts componentInstance, resourceInstance, and requiredMemory. The requiredMemory named part represents the required memory of the component that is referred by the componentInstance named part, if it is allocated to the ECU that is referred by the resourceInstance named part.

Each 2-tuple represents a constraint whose left-hand side (lhs) has to be smaller than or equal to the right-hand side (rhs). The left-hand side is the sum over the referred 3-tuples. A 3-tuple

contributes its requiredMemory value to sum, if its referred componentInstance is allocated to its referred resourceInstance, and 0 otherwise. The right-hand side is given by the 2-tuple's availableMemory named part.

```
1 constraint requiredResource maxMemoryConsumption {
2 lhs requiredMemory;
3 rhs availableMemory;
4 descriptors (componentInstance, resourceInstance);
5 ocl self.maxMemoryConsumption(); }
```

Listing 4.4: A Constraint of the Kind requiredResource

4.2 Extended MechatronicUML Development Process

In Work Package 2 of Amalthea4public, we integrate App4mc and MechatronicUML. This enables MechatronicUML to be used as a modeling frontend within the Amalthea ecosystem. Here, we focus on the extension of the existing MechatronicUML allocation. This extension enables the developer to specify a multi-core scheduling not only for single MechatronicUML software components but for *ECU networks*, explicitly taking communication- and hard real-time requirements into account. To deploy the software specified in the platform-independent model to a multi-core environment, we extend the subprocess of process step 4 "Design Platform-Specific Software" (cf. Figure 4.1). In the following, we call this process Platform Specific Modeling (PSM) process.

In the current PSM process, a task is created for each component instance of the PIM and allocated to an ECU. Timing properties are not considered and task properties like period, deadline, and Worst Case Execution Time (WCET) are derived manually. However, since we are aiming for deploying the software to a multi-core platform consisting of an ECU network, additional steps are needed for the allocation of software components and runnables. In this section, we give an overview of the extended development process that is used to determine a multi-core scheduling for MechatronicUML. We refer to [28] for a detailed description of the concepts we sketch here.

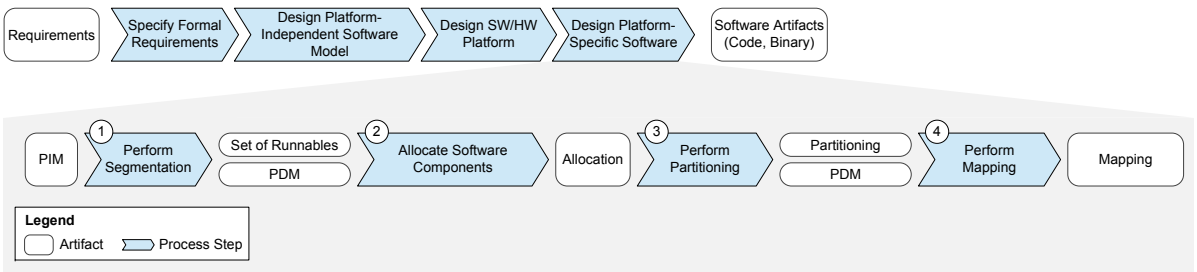

Figure 4.2: Extended MechatronicUML PSM Process (based on: [28])

Figure 4.2 shows all steps of the extended PSM process. Steps 1, 3, and 4 are new process steps. Step 2 extends an existing one. We assume that a correct PIM and PDM are already defined at this point of development and are used as input artifacts for the presented steps. In the following, we describe each process step, the order of the steps in the process, and their

input and output artifacts. The embedding of the presented development process in a complete systems engineering development process is presented in [2].

At first, in **Step 1**, the software is separated into independently executable pieces that correspond to AUTOSAR Runnables. We call this step *Perform Segmentation*. For each runnable, runnable properties are defined (WCET, period, and deadline). Additionally, dependencies and constraints between the runnables are defined, *e.g.*, accesses to shared variables. The output artifact of this step is a set of runnables with defined runnable properties.

In **Step 2** the software is allocated to ECUs, *i.e.*, we define which runnables are executed on which ECU. This step is called *Allocate Software Components*. Hence, the input for this step is a set of runnables (produced by the segmentation) and the PDM. Since an allocation can affect the communication latencies, we ensure in this step that all Quality of Service (QoS) assumptions of MechatronicUMLs Real-time Coordination Protocols are respected. The output artifact of this step is a *runnable allocation* that describes the allocation of runnables to different ECUs of an ECU network.

In **Step 3**, we define for each of the ECUs which runnables can be executed by one task. This step is called *Perform Partitioning*. The output artifact of this step is a partitioning for each ECU, *i.e.*, a set of tasks per ECU. The partitioning per ECU is beeing calculated by means of existing algorithms available in Amalthea.

In **Step 4**, we define for each ECU which task is executed on which ECU core. This step is called *Perform Mapping*. The input for this step is the partitioning for each ECU. In the mapping, each task gets assigned to an ECU core that will execute this task at runtime. The mapping has to respect dependencies between the tasks and their runnables. The output of this step is a mapping of tasks to ECU cores. Again, we use existing algorithms from Amalthea to perform the mapping.

Finally, the remaining steps of the MechatronicUML PSM process are applied, *i.e.*, steps for *Platform Mapping*, *Code Generation*, and *Analysis*. In the *platform mapping*, hardware abstractions in the PIM, like continuous components, get enriched by concrete platform dependent parts, *e.g.*, concrete Application Programming Interface (API) calls for sensors and actuators. Afterwards, the newly created PSM, provided for each ECU, is transformed into source code for the target platform and compiled to an executable. Finally, the software can be executed and logged to analyze the execution on the target platform. For this, traces can be used for further analyses of the software system, *e.g.*, by using methods of Amalthea4public. In the next sections, we present more details on step 2. Please refer to [1] for a detailed description of step 1.

4.3 Partitioning and Mapping for ECU Networks

In MechatronicUML, software components do not run in isolation but are connected to other component instances for communication. Since we consider software with hard real-time requirements, this communication has to fulfill hard real-time requirements as well. In MechatronicUML, Real-time Coordination Protocols (RTCPs) are used on PIM level to define QoS assumptions for communication between interacting components. These QoS assumptions are used to verify the fulfillment of these hard real-time requirements using model checking. Hence, the mapping has to ensure these assumptions, especially if more than one ECU is available in the system.

An example scenario for a distributed, cooperative, and safety-critical Intelligent Technical

System (ITS) is the autonomous overtaking of vehicles, *e.g.*, cars. Here, we re-use such a scenario from [28] as a running example for explanation of the concepts. We use MECHATRONICUML models for an overtaking scenario where an approaching vehicle is considered. The scenario consists of *three vehicles* and a *section control*. Figure 4.3 shows a sketch of this scenario.

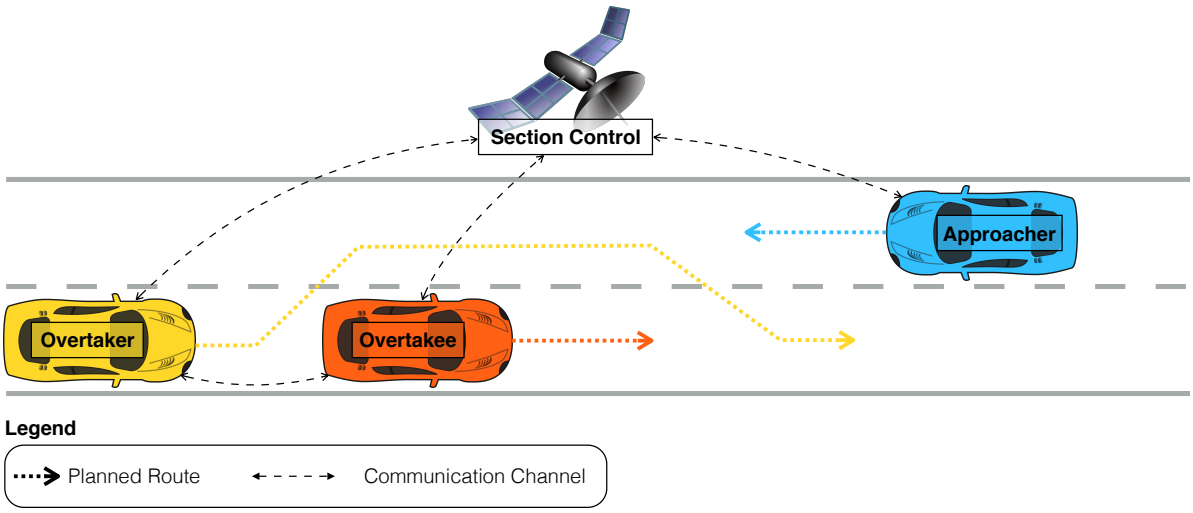

Figure 4.3: Scenario for Overtaking With Approacher (from [28])

Two vehicles, `Overtaker` and `Overtakee` drive on one lane in the same direction. On the other lane, a third vehicle (`Approacher`) is driving in the opposing direction. Each vehicle communicates with the `Section Control` and informs about its current position on the lane. Thus, the `Section Control` knows the position of each vehicle. The vehicles themselves do only know their own position.

The `Overtaker` drives with higher speed than the `Overtakee` and when the `Overtaker` gets to a specific distance to the `Overtakee`, it adapts its speed and starts to communicate with the `Overtakee` and the `Section Control` to initiate an overtaking action. It waits for acknowledgments of both communication partners before starting the overtaking. The `Section Control` has to confirm that the distance to the `Approacher` is large enough. The `Overtakee` can accept the overtaking request and confirms that it will not accelerate during the overtaking. Here, we focus on the `Overtaker`. Figure 4.4 shows its Component Instance Configuration (CIC).

The CIC of the `Overtaker` consists of seven atomic components. The components `OvertakerDriver` and `OvertakerCommunicator` define Real-time State Charts (RTSCs) as behavior and, therefore, have timing requirements that have to be considered in segmentation, partitioning, and mapping. Additionally, the communication of these components is defined by a so-called RTCP `Delegate`. A RTCP defines QoS assumptions that have to be considered when deriving a multi-core scheduling, *e.g.*, an upper bound for the communication time. Furthermore, we assume that the hardware for each vehicle consists of two multi-core ECUs.

In step 2 of the extended PSM process, we extend the existing allocation with an approach that enables the developer to specify a multi-core scheduling for an *ECU network* that explicitly takes communication requirements in hard real-time systems into account. The input for this step is a set of runnables with defined runnable properties produced by the segmentation in step 1. First, we allocate runnables to ECUs. Since all runnables that belong to one

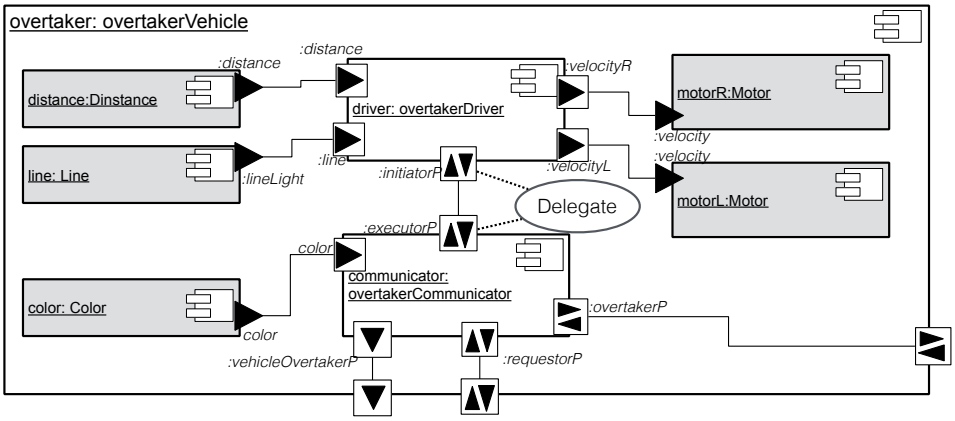

Figure 4.4: Component Instance Configuration of the Overtaker (from [28])

component instances are highly dependend, we propose to allocate all runnables of a component instance to the same ECU. Hence, in this step, we allocate component instances to ECUs with respect to the runnable properties and to the communication requirements. At this point of development, we cannot apply a full schedulability analysis. Nevertheless, it is important that at least necessary conditions for schedulability are respected in the allocation step. In our approach, we ensure that the processing capacity of the ECU is not exceeded by the the processing requirements for the allocated runnables, *i.e.*, we discard allocations that are not schedulable already in this step. Additionally, we ensure that the real-time requirements for the communication are respected in the allocation. These two features are provided by an extension of the ASL. We extend the ASL by providing two new ASL constraints that can be used by the developer: *validUtilization* and *maxDelay4RTCPs*. *validUtilization* provides an ASL constraint to ensure a necessary *Schedulability-condition* for runnables. *maxDelay4RTCPs* provides an ASL constraint to satisfy the max-delay defined in RTCPs during the allocation. In principle, the constraint ensures that the amount of computing time of the executed software does not exceed the processing capacity of each ECU.

Condition for Schedulability Here, we present a necessary *Schedulability*-condition for runnables. At his point in time, we cannot apply a full schedulability analysis for the software, because we only have specified runnables, but do not know anything about possible tasks. Tasks will be determined during implementation by the Amalthea partitioning algorithm. Nevertheless, we can restrict the allocation regarding a necessary condition for schedulability: The amount of computing time of the executed software must not exceed the processing capacity of the ECU. We define the processing capacity of each ECU core as 1. Consequently, the processing capacity of each ECU C_{ECU} is defined as

$$C_{ECU} = |ECUCores| \tag{4.1}$$

Each runnable has a defined Period π and a WCET. Note that the WCET does depend on the executing ECU and may vary if the runnable is allocated to different WCETs. Let $WCET_{Runnable,ECU}$ be the WCET for a runnable and a specific ECU. We define the utilization factor of a runnable $U_{Runnable}$ for a specific ECU as

$$U_{Runnable} = \frac{WCET_{Runnable,ECU}}{\pi_{Runnable}} \tag{4.2}$$

If the sum of the utilization factors of all runnables exceeds the processing capacity of the ECU, it is impossible to find a scheduling for a given set of runnables (independent from the resulting tasks). If this sum equals the processing capacity of the ECU, the resulting scheduling would have to execute all runnables without any idling time. Since this is not realistic for complex systems, we define as a necessary condition that this sum has to be *less* than the processing capacity of the ECU.

$$\sum_{r \in Runnables(ECU)} U_r < C_{ECU} \tag{4.3}$$

Condition for Communication Latency A RTCP defines a contract between communication partners. Each communication partner has to refine the behavior of a role, defined by the RTCP. In this section, we present an approach how the max-delay defined in RTCPs can be ensured during the allocation.

A RTCP provides QoS assumptions, like the min/max delay for the communication. Furthermore, it refers to RTSCs that define the behavior of the roles and are refined for the RTSCs of each port that implements the specific role. We have to consider each instance of RTCPs in the component instance configuration separately. Figure 4.5 shows the RTCP that is applied to component `overtakerDriver` and `overtakerCommunicator` in our example.

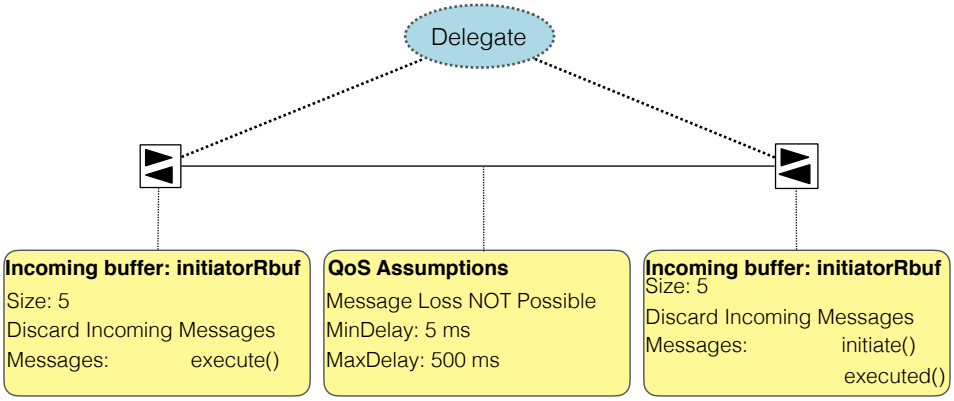

Figure 4.5: RTCP Delegate with QoS Assumptions (from [28])

The RTCP defines a `min delay` of `5 ms` and a `max delay` of `500 ms`. Thus, the message has to be transmitted within this time interval. Otherwise, the message will be dropped, *e.g.*, by the middleware of the system. For our approach, we focus on the max delay. If the message arrives before `min delay`, we assume that the middleware will retain the message until it can be delivered. Delivering a message relies basically on time for generating and sending the message, transmitting it from sender to receiver, and queuing it until the receiving process recognizes the message. The max-delay defines the maximum time span we have for sending, transmitting, and receiving the message. Thus, we have to consider

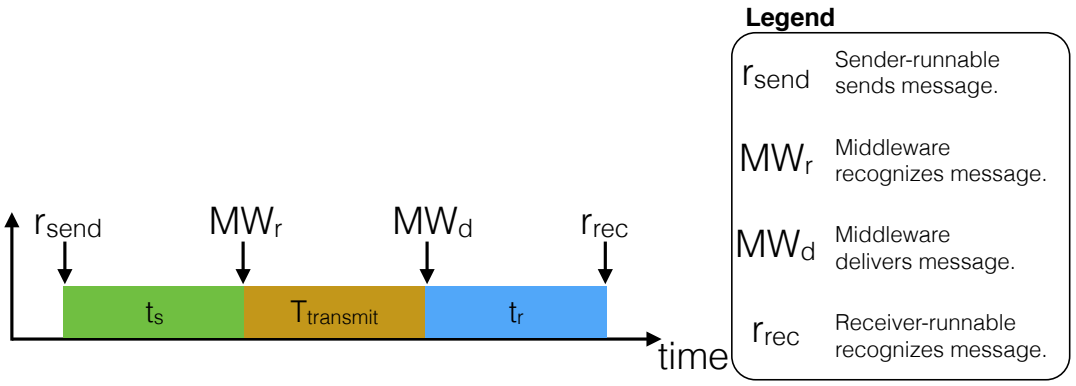

Figure 4.6: Transmitting a Message can be Separated into Three Time Slots (from [28])

these time frames for deriving the condition. Figure 4.6 shows these time frames in an exemplary schedule.

At r_{send} the message is sent by the sender runnable. At MW_r the middleware recognizes that a message has been sent. We call the interval between these points as *Time for Sending* and denote it by t_s. After that, the message is transmitted to the target and is put into the corresponding message buffer of the component instance at MW_d. We call the interval between MW_r and MW_d *Time for Transmitting* and denote it by $t_{transmit}$. At r_{rec}, the receiver runnable evaluates its transitions and recognizes the new message. We call the interval between MW_d and r_{rec} *Time for Receiving* and denote it by t_r.

Using the upper bound values for t_s, $t_{transmit}$, and t_r, we can express a constraint implied by the QoS-assumptions of the RTCP (we refer to [28, Section 5.2.2] on how these upper bounds are derived). Let T_{RTCP} be the max-delay value of the RTCP. Since T_{RTCP} specifies the upper bound for sending, transmitting, and receiving the message, the sum of these three values has to be smaller than the max-delay. Consequently, for each RTCP that is used in the system, the following inequation has to hold:

$$t_s + t_{transmit} + t_r \leq T_{RTCP} \tag{4.4}$$

As $t_{transmit}$ depends on source and target ECU, it depends on the allocation of component instances to ECUs, if inequation 4.4 can be satisfied or not for each RTCP. Thus, they have to be respected in the allocation step in the development process.

Example Here, we show an example for applying the *maxDelay4RTCPs* constraint to our example. For this, we assume that the segmentation of the CIC is already finished and therefore, we have a defined CIC with references to dedicated runnables. We assume that the segmenation strategy *One Runnable Per Region* (cf. [28, Section 4.2.1.3]) is applied for discrete component instances. Figure 4.7 shows a sketch of the CIC for component `overtakerVehicle` and the resulting runnables.

As example, we apply the communication latency condition (cf. inequation 4.4) in the allocation. Hence, we focus on the runnables that communicate and, therefore, contain port behavior, *i.e.*, runnables `initiatorP` and `executorP`.

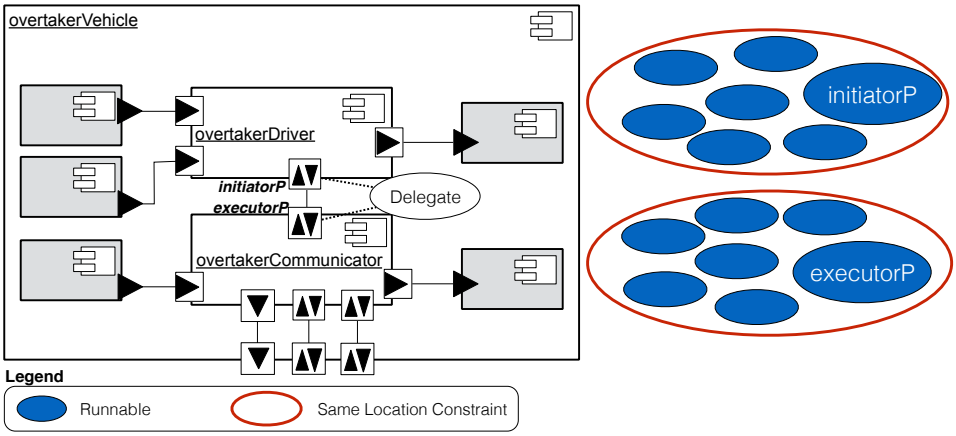

Figure 4.7: CIC for `overtakerVehicle` and the resulting Runnables (from [28])

Additionally, we assume that the PDM is defined and accessible by the ASL. The developer may also define ASL-constraints to ensure important allocation requirements, *e.g.*, that all component instances of the overtaker are allocated to the ECUs of one vehicle using the *sameLocation* constraint of the ASL. Additionally, we assume that all continuous component instances and the connected discrete component instances are allocated to the same ECU. In Figure 4.7, this is indicated by the red lines.

Additionally, the RTCP `Delegate` is used between the ports `initiatorP` and `executorP`, which defines a max delay of $500ms$ (cf. Figure 4.5). Let us assume that the developer added the ASL constraint for communication latency condition (cf. inequation 4.4) to the allocation specification. Table 4.1 shows exemplary values for the period of sender runnable and receiver runnable that are used in the constraints.

Table 4.1: Exemplary Runnable Properties for the Example-CIC (from [28])

Runnable	Period π	t_s	t_r
initiatorP-Runnable	180 ms	180 ms	360 ms
executorP-Runnable	100 ms	100 ms	200 ms

Thus, we have $t_s = \pi_{initiatorP} = 180ms$ and $t_r = 2 * \pi_{executorP} = 200ms$. Furthermore, we assume that $T_{transmit}$ between ECU e_1 and ECU $e2$ has the constant value of $150ms$. $T_{transmit}$ for the communication on the same ECU has the constant value of $0ms$, *e.g.*, when global variables are used for implementation. However, if more ECUs are involved and, therefore, $T_{transmit}$ may vary between different ECU-pairs, $T_{transmit}$ can be annotated to the used communication channel.

In the following, we show the four ILP-Constraints resulting from the ASL-constraint for the RTCP `Delegate`. For better readability, we focus on one direction of the communication, *i.e.*, sending a message from `initiatorP` to `executorP`. Additionally, the result for each inequation is shown using the values from Table 4.1.

$$(t_{s,iniP}+T_{transmit(e1,e1)}+t_{r,exeP}) \quad *x_{CI(iniatorP),e1,CI(executorP),e1} \leq Delegate_{maxDelay}$$
$$(180+0+200) \quad *x_{CI(iniatorP),e1,CI(executorP),e1} \leq 500 \tag{4.5}$$

$$(t_{s,iniP}+T_{transmit(e1,e2)}+t_{r,exeP}) \quad *x_{CI(iniatorP),e1,CI(executorP),e2} \leq Delegate_{maxDelay}$$
$$(180+150+200) \quad *x_{CI(iniatorP),e1,CI(executorP),e2} \leq 500 \tag{4.6}$$

$$(t_{s,iniP}+T_{transmit(e2,e1)}+t_{r,exeP}) \quad *x_{CI(iniatorP),e2,CI(executorP),e1} \leq Delegate_{maxDelay}$$
$$(180+150+200) \quad *x_{CI(iniatorP),e2,CI(executorP),e1} \leq 500 \tag{4.7}$$

$$(t_{s,iniP}+T_{transmit(e2,e2)}+t_{r,exeP}) \quad *x_{CI(iniatorP),e2,CI(executorP),e2} \leq Delegate_{maxDelay}$$
$$(180+0+200) \quad *x_{CI(iniatorP),e2,CI(executorP),e2} \leq 500 \tag{4.8}$$

The results show, that the inequations 4.5 - 4.8 can only be fulfilled, if both runnables are executed on the same ECU. Consequently, the QoS assumption of the RTCP can only be satisfied, if both component instances are allocated to the same ECU. Since we specified that all continuous component instances have to be allocated to the same ECU as the connected component instance, the resulting allocation is, that all runnables have to be allocated to the same ECU. Thus, the second ECU is not used. This might be a problem, if additional ASL-constraints are used that restrict the allocation of all component instances to the same ECU, *e.g.*, when the constraint for the utilization factor cannot be fulfilled by this allocation. Hence, the developer should check all runnable properties and the RTCP for possible changes to relax the conditions for the ILP.

For each ECU, the resulting runnable sets can then be processed further by the Partitioning- and Mapping-Step. At the end, a mapping that contains tasks is provided to execute the software on each ECU.

Specifying Allocation Constraints We have to find an allocation, for which the conditions 4.3 and 4.4 hold. Thus, we have to specify constraints for each possible combination of ECUs and runnables. An advantage of using the ASL is, that many of the resulting constraints for these conditions can be expressed by only one ASL-constraint. The ASL defines four different kinds of constraints. We reuse the constraint kind *RequiredResource* for the new constraints since it provides the needed functionality to generate the ILP-constraints. Using the extended context-model for the ASL, each of both inequations can be specified in one ASL constraint that hides the concrete implementation from the developer. Listing 4.5 shows the concrete call of the ASL-constraints.

```
1 --ensure RTCP max delays in allocation
2 constraint requiredResource maxDelay4RTCPs {
3   ...
4   ocl self.RTCPMaxDelayConstraints();
5 }
6
7 --ensure necessary condition for scheduling
```

```
8  constraint requiredResource validUtilization{
9    ...
10   ocl self.validUtilizationFactors();
11 }
```

Listing 4.5: ASL constraint specification for the new constraints.

The developer has to specify a requiredResource constraint for each of both conditions. We ommit some ASL specific parts of the constraints, indicated by three dots. The concrete OCL code to determine all constraints is called by one OCL-Operation for each condition. Hence, the developer does not need any domain-knowledge to use our approach in the allocation. These ASL-constraints are translated to corresponding ILP-constraints during the allocation automatically. Next, we show the expected number of ILP-constraints shortly and how they can be derived for both conditions.

In general, inequation 4.4 can directly be translated into an ILP-constraint. For this, we have to add an ILP-decision-variable to the constraint. Let x_{r_s,e_x,r_r,e_y} be the decision-variable that is 1, if runnable r_s is allocated to ECU e_x and runnable r_r is allocated to ECU e_y. In all other cases it is 0. Then, we can define the following ILP-constraint:

$$(t_s + t_{transmit(e_x,e_y)} + t_r) * x_{r_s,e_x,r_r,e_y} \leq T_{RTCP} \tag{4.9}$$

where t_s is the *Time for Sending*, t_r is the *Time for Receiving*, and $t_{transmit(e_x,e_y)}$ the time for sending a message from ECU e_x to ECU e_y.

Obviously, this inequation has to hold for every combination of the RTCPs with every possible pair of ECUs. Thus, we have to specify $|ECU|^2 * |RTCP|$ many constraints, because we have to consider all possible pairs of ECUs for each RTCPs.

Furthermore, inequation 4.3 can also be translated directly into an ILP-constraint by adding a decision-variable to the inequation. Let $x_{r,e}$ be this decision-variable that is 1, if runnable r is allocated to ECU e. In all other cases, it is 0. Let C_e be the processing capacity of ECU e. Then, we define e the following constraint for each ECU.

$$\sum\nolimits_{r \in Runnables} (U_r * x_{r,e}) < C_e \tag{4.10}$$

Thus, we have to define $|ECU|$ many ILP-constraints. Note, that U_r is not a static value for each runnable but does depend on the executing ECU (cf. equation 4.2).

For both new inequations, we have to define $|ECU|^2 * |RTCP| + |ECU|$ many ILP-constraints in total that can be expressed by only two ASL-constraints as shown in Listing 4.5.

5 Safety Considerations in Partitioning and Mapping

5.1 Motivation

In the development of software for multi-core ECUs, various challenges regarding functional safety aspects as, e.g., defined by ISO26262 [33] are given. The main challenge is to ensure freedom from interference as defined in ISO26262, Part 6 [36, 7.4.11] between different software partitions. In current developments this is to the best of our knowledge most often done manually. That is, the decision to move a particular software partition to a particular ECU is based on the experience of the development team aided by using development tools. In case of uncertainty within the development team that might arise due to the lack of tool support, usually *safe* decisions are drawn. Typically this means, that in the case of uncertainty whether software partition A influences software partition B such that a given goal is violated, the decision will be to use, e.g., different ECUs for each of the software partitions when dealing with system where only one core is used for the computation of the E/E function.

For multi-core systems the challenge becomes even larger since effects like, for example, race conditions, have to be considered. Although these are challenges that effect the overall development of software for multi-core systems, we focus here on the challenges that are imposed by standards on functional safety that have to be respected in the development of automotive E/E systems. In particular, the applicable standard is the ISO26262 [33] and for the development of software Part 6 of the ISO norm has to be considered.

The safety life-cycle prescribes that *safety goals* will be formulated. Based on the *safety goals* the *functional safety concept* will be defined in terms of *functional safety requirements* that are allocated to architectural elements of the item under development. In a next development step, the *functional safety concept* will be refined to the so-called *technical safety requirements* on system level, see Part 4 of ISO26262 [35]. In particular, the *technical safety requirements* specify the system's response to stimuli that affect the achievement of *safety goals* [35, 4.6.4.2.1]. Subsequently, the system design and the *technical safety concept* will be developed. Here, the allocation of *technical safety requirements* to system design elements is one of the main development activities in order to fulfill the requirements of ISO26262 [35, 7.4.1.2]. In subsequent development steps, the *technical safety concept* is used to derive both hardware and software safety requirements. As given in [35, 6.2]: "the software safety requirements have to consider the constraints that are given by the hardware and the impact of these constraints on the software". The software safety requirements have to consider, e.g., timing constraints. When these software safety requirements are allocated to software architectural design elements, as forced by ISO26262 Part6 [36, 7.4.9], it has to be ensured that freedom from interference is fulfilled. When realizing freedom from interference with methods of software partitioning, various requirements have to be considered. For example, (1) neither code nor data shall be changed by software of another software partition, and (2) the use of shared resources by a software partition must not affect another software partition.

Based on the above stated challenges, a method would be useful that allows to attach software runnables with a sufficient kind of information on the (functional, technical) safety concept such that partitioning and mapping can be done algorithmically. In terms of APP4MC, this could be done using *Affinity Constraints* as described in Section 2.2.4 and Property Constraints as described in Section 8.2 of AMALTHEA Deliverable 3.3 [5].

5.2 Functional-safety enhancement for partitioning and mapping

In order to realize a method for partitioning and mapping of software runnables, it will not be enough to consider the Automotive Safety Integrity Level (ASIL), as given in ISO26262 Part3 [34, 7.4], alone.Additional information as, e.g., timing information, label accesses and the usage of communication interfaces such as CAN or FlexRay have to be considered.

In the following, an example case is illustrated, that contains `Runnable Pairing-`, `Runnable Separation-` and `Property Constraints` in order to support functional safety aspects during partitioning and mapping.

In order to define the scope of this example, two prerequisites have to be introduced: Firstly, we introduce the term "environment" as all the underlying hardware and software under which a runnable is executed. We require that the environment in which the components of interest shall be executed has the necessary functionality and interfaces not only from a computational point of view but also in terms of functional safety. This means, that the hardware and runtime environment are equipped with functional safety measures to allow the execution of software up to a certain safety level. Hence, we can expect a all needed safety infrastructure to be in place if we assign a runnable of a certain ASIL level to a software and/or hardware partition that is qualified with the same or a higher ASIL level as the runnable itself. We will call this an "infrastructure requirement" in the further reading of this section.

Secondly, we assume that the runnables to be distributed have been developed according to functional safety standards. Frameworks like AUTOSAR encourage developers to do so and provide respective libraries to aid functional safety compliant development but do not necessarily enforce such behaviour [20]. In the worst case this could mean that software components circumvent the safety measures of the underlying systems and by this compromise the safety concept. However, to realistically come to an automated distribution and mapping of runnables to ECUs and cores therein we have to be able to treat these runnables as black boxes with certain standardized interfaces and requirements and functional safety implications that can be derived from these black box specifications. Therefore we rely on the assumption that runnables do comply to functional safety standards and do not work around such mechanisms.

5.3 Example case

The example case illustrates the functional safety considerations of a basic *adaptive cruise control* (ACC) system and the constraints that these safety considerations impose on the software components. The architecture of the software runnables and their relations within the fictional ACC software system are depicted in Figure 5.1 while important aspects will be visualized in the zoomed views in Figures 5.2 and 5.3.

As a result of the safety analysis of the ACC system the system has to comply to ASIL level *C* and should have an overall Worst Case Response Time (WCRT) of 250 ms, *i.e.*, the time between an external stimulus and the system response should be less than the WCRT. In

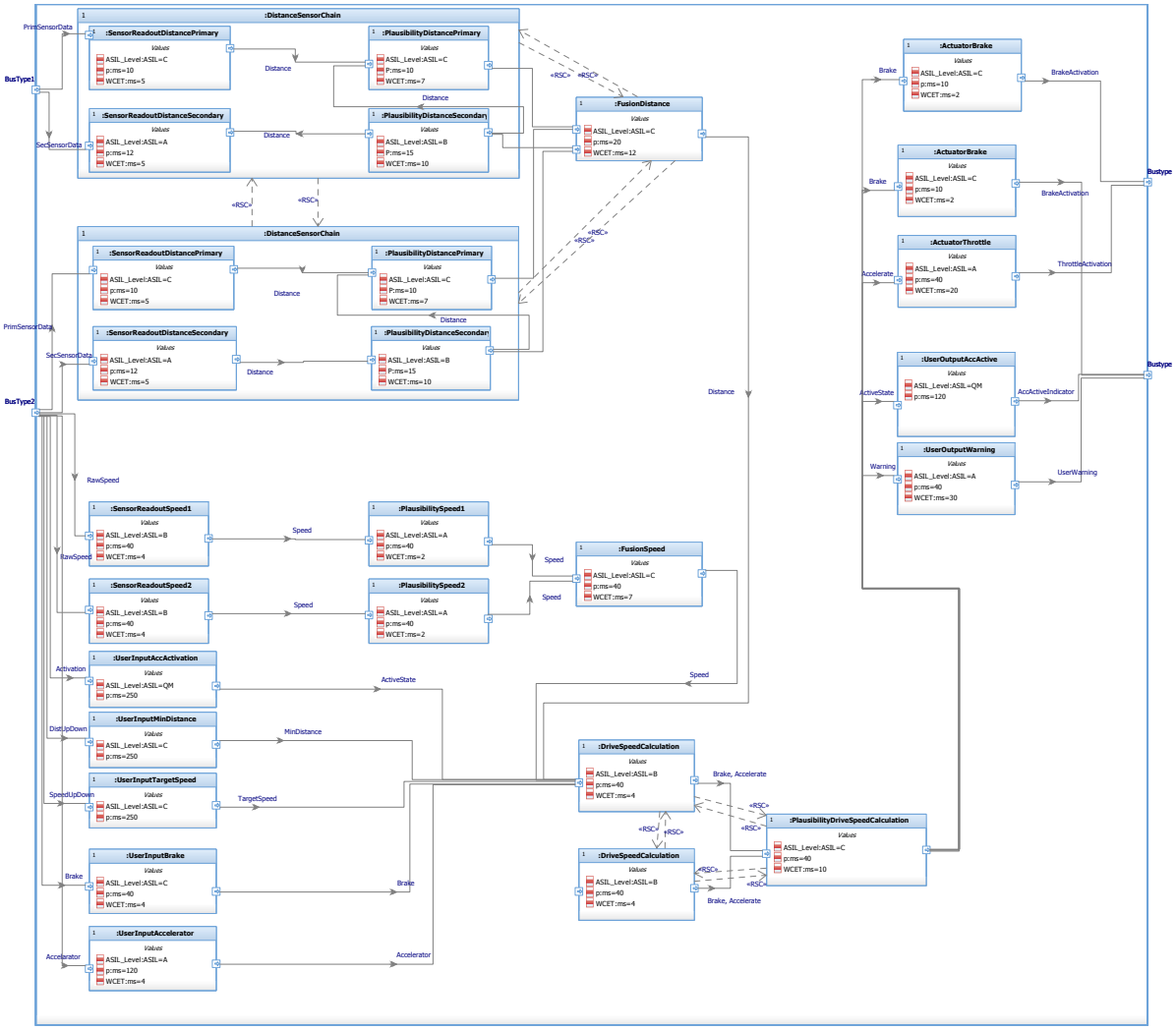

Figure 5.1: Architectural model of the example ACC

the following work process these safety requirements for the whole system have been decomposed and translated into safety requirements for every single component within the system.

5.3.1 Functional safety relevant information attached to runnables

Each runnable can have the following safety relevant information attached which have been derived during the development and implementation of the safety concept:

- its ASIL level,
- the WCET, which is the maximum time a runnable may take between being called and terminating successfully,
- the execution period p, *i.e.*, the time intervals in which the runnable must be executed, and
- the *qualified* dependencies of the runnable to other runnables.

The first three items are indicated as attributes of the respective blocks while the dependencies are visualized by the connections between single runnables.

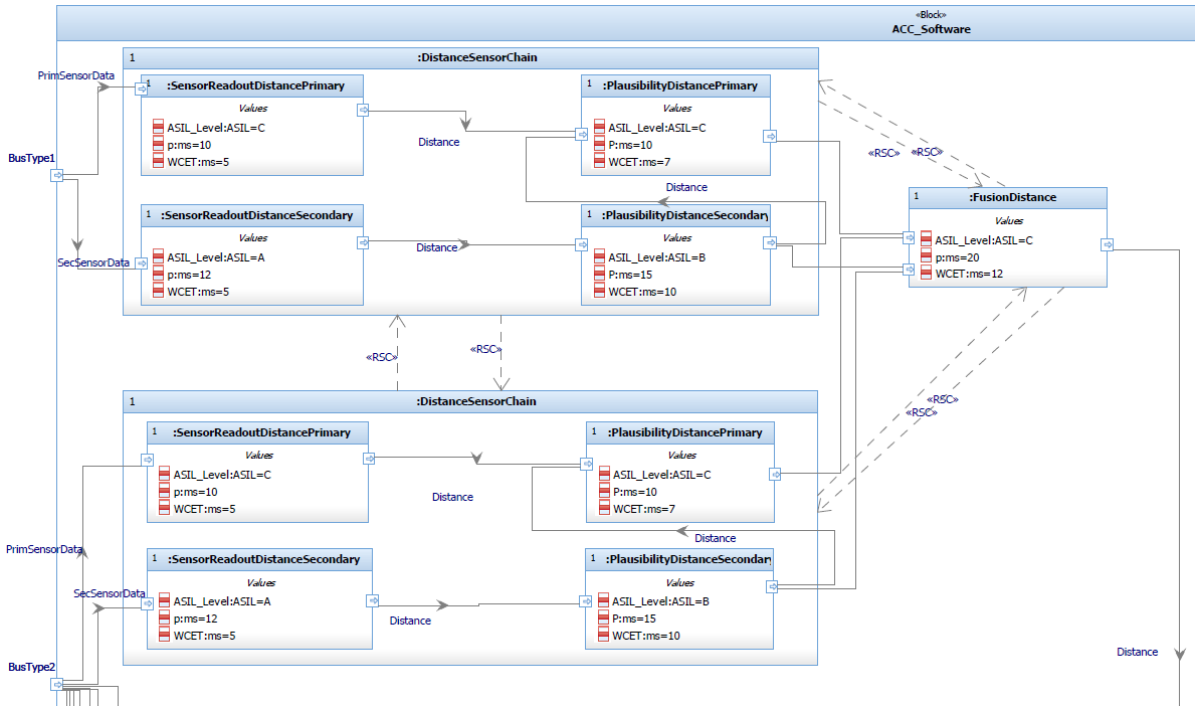

Figure 5.2: Detail view of the distance sensor part in the ACC model.

ASIL level

The ASIL level imposes several implications which are mostly infrastructure requirements as defined in Section 5.2 for the distribution and mapping of runnables to comply to the concept of freedom from interference. First and foremost a runnable of a certain ASIL level is only allowed to run in an environment that has been developed under the same or a higher ASIL.

Secondly, the concept of freedom from interference demands that runnables of different ASIL are not allowed to have *direct* access to the same resources like memory or labels. Especially, it has to be prevented that a runnable of lower safety level can compromise the execution of a runnable of higher safety level. Such a situation can be seen in Figure 5.2 between the components "SensorReadoutDistanceSecondary" (ASIL A) and "PlausibilityDistanceSecondary" (ASIL B) as well as "PlausibilityDistanceSecondary" and "PlausibilityDistancePrimary" (ASIL C). In such situations the respective runnables have to be put in different partitions of the environment and the safety mechanisms provided by the environment have to ensure freedom from interference by supervising the data flows between them.

Worst Case Execution Time and execution period

The WCET relates to the longer WCRT that is used on a higher level like a group of software components by the following relationship: The total WCRT of a group of software components C_i that are executed sequentially and connected to each other via the communication paths j with signaling time constants t_j is $\mathrm{WCRT}_t = \sum_i WCET_i + \sum_j t_j$. Where a component contains several components or component groups that are executed in parallel the WCRT of the whole component is just defined by the path(s) in the parallel execution with the longest WCRT. By these two rules it is possible break the WCRT of a whole system or software components down to a WCET for every single runnable.

As already the ASIL level the WCET and execution period impose mainly an infrastructure requirement since it has to be ensured that the partition to execute a certain runnable provides enough resources to run it within the required time frame and frequency.

Dependencies

For the qualification of the dependency between runnables we have two dimensions: safety-criticality and timing-criticality. Hence, the following four possibilities exist for a qualified dependency:

1. safety-critical and timing-critical,
2. safety-critical and not timing-critical,
3. not safety-critical and timing-critical.
4. not safety-critical and not timing-critical,

However, the cases 3 and 4 can be treated equally because the timing-dependency on non-safety critical dependent runnables is not of importance for the partitioning and mapping of those runnables.

The qualification of a dependency can be derived by looking at the ends of a connector between two blocks: Criticality in either of the two dimensions is given if both ends are have a respective criticality property, *i.e.*, both have an ASIL level higher than "QM" and/or have a WCET attached. If no or only one end of the connection has such a property attached, the dependency can be considered not critical in the respective dimension. Examples of the cases 1, 2, and 4 can be seen in Figure 5.3: With respect to the block "DriveSpeedCalculation" (ASIL B and with a WCET), the connection to "UseInputBrake" (ASIL and WCET) is safety and timing critical (case 1), the connection to "UserInputTargetSpeed" (ASIL but no WCET) is

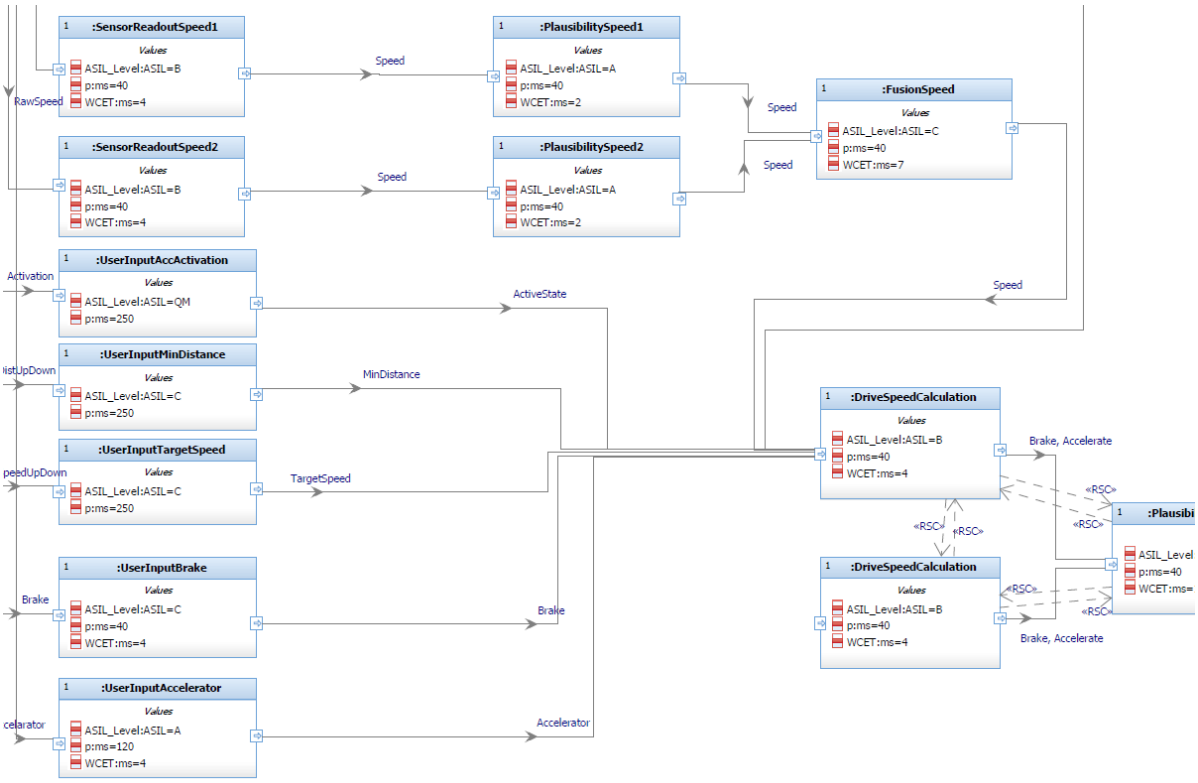

Figure 5.3: Detail view of the user input section of the ACC model.

only safety critical (case 2), while the connection to "UserInputAccActivation" (ASIL QM and no WCET) is neither safety nor timing critical (case 4).

In addition to the three cases above, we define another class of dependency between runnables which we call *resource safety-critical* (RSC). Two runnables are RSC with respect to a resource A, if they are not allowed to share resource A. Here, resource A means especially resources that are outside of the direct control of the respective software system, *i.e.*, components of the environment like access to memory partitions, processor cores, or communication interfaces. Resources that are within the control of the software system such as access to shared labels where some runnables have write and and another runnables have read access should already be covered by safe programming concepts like a client-server or sender-receiver approach. That way they can be handled safely by the safety measures of the underlying operating system if runnables are separated across operating or hardware resource borders that are not necessary by safety considerations but only have been introduced by the distribution of the runnables to different ressources under other than safety-related criteria.

RSC dependencies typically come into play where the safety concept requires a redundant design to minimize the risk of failure. Such RSC dependencies can be seen in Figure 5.2 between the two big "DistanceSensorChain" blocks and the "FusionDistance" block and in Figure 5.3 between the "DriveSpeedCalculation" blocks. Unless there exists already an explicit RSC dependency definition concerning certain resources from the safety concept – like the two "DistanceSensorChain" blocks in Figure 5.2 which by design get their inputs from two redundant bus types ("BusType" ports on the left edge of Figure 5.2) – it is yet to be checked whether a heuristic can be derived to determine by which resources a separation should be done.

5.3.2 Relation to general mapping and partition constraints

For this ACC exmple case, various requirements arise to the partitioning and mapping of runnables. These requirements can be fulfilled by the methods introduced in Chapters 2 and 3. The concrete application of these methods to the previously introduced example, and more generic to safety-standard related developments, will be discussed in the upcoming deliverable, i. e., D2.3, of WP2.

6 Resource Management

This chapter addresses resource management in embedded and concurrent systems. Partitioning, mapping, and resource protocols influence each other such that different combinations should be investigated and assessed. This assessment can be done by analyzing traces in order to check for busy waiting periods (the time a resource keeps spinning until a locked resource becomes available).

Shared memory must be managed by mechanisms, algorithms or protocols to support error-free and simultaneously effective utilization of shared resources. Usually, a semaphore is used to lock a resource, perform an action, and release (unlock) the resource afterwards. The problem of deadlocks, spinlocks, and mutual exclusion as well as regarding concepts of AUTOSAR were already outlined in Deliverable D2.1 [8].

6.1 Protocols

The protocols presented in this chapter aim at the automatic and efficient management of accessing shared resources while considering timing, precedence or resource constraints. Timing constraints can be activation, completion (deadlines) or jitter. Precedence constraints usually impose execution orders whereas resource constraints ensure a synchronized access to mutual exclusive resources. Gomes *et al.* [29] describe briefly various protocols in the domain of resource synchronization protocols.

The most common way of lock implementations are semaphores consisting of:

1. a counter,
2. a queue,
3. a pointer to the next semaphore control block,
4. and a holder .

The next sections describe the evolution of basic protocols from the early beginning of NPP (Section 6.1.1) to a complex and very recent approach PWLP (Section 6.1.12). Basic ideas, extensions, benefits, and references are given. The list does not intend to be a cross-domain complete enumeration but rather gives an impression of shared resource management protocol's evolution.

6.1.1 NPP

The Non Preemption Protocol (NPP) gives the 'caller' (in terms of App4mc the runnable that accesses the shared label) the highest priority such that it will not be preempted. Upon releasing the resource, the 'caller' resets its priority to its initial value. However, this most simple method results in low priority tasks blocking higher priority tasks resulting in possible missed deadlines but avoids deadlocks.

6.1.2 BIP / PIP

The Priority Inheritance Protocol (PIP) [45] or Basic Priority Inheritance Protocol (BIP) sets the lock-holder's (L) priority to the priority of a higher priority task (H), if H's lock request is rejected since L is holding it. In other words, the blocked task's priority is inherited to the blocking task. The disadvantage of BIP / PIP is that it is subject to potential deadlocks and chained blocking occurs with many preemptions. In contrast to PCP (Section 6.1.4), priority inheritance does not require a priori resource ceiling value calculations.

6.1.3 HLP

The Highest Locker Protocol introduces a ceiling value to locks (semaphores) that is the highest priority of the tasks that access the lock (statically calculated offline). This approach is deadlock free since a task holding a lock runs at the same priority as any other tasks that want to access the lock, such that the other tasks will no preempt the lock holding task. The 'Holder' in the semaphore structure (Section 6.1 item 4) is replaced by the ceiling priority for HLP [17].

6.1.4 PCP

The Priority Ceiling Protocol [45] is a combination of HLP and BIP. A blocking task (the task holding a lock) increases its priority only if a higher priority task tries to acquire the lock (the blocking task's priority is set to the higher priority task's priority). PCP results in bounded priority inversion, deadlock free results, blocking number = 1, and a better response time for high priority tasks.

6.1.5 SRP

The Stack Resource Policy (SRP) protocol [11] extends PCP by "(1) unifying the treatment of stack, reader-writer, and multiunit resources, and binary semaphores; (2) applying directly to some dynamic scheduling policies, including EDF, as well as to static priority policies; (3) with EDF scheduling, supporting a stronger schedulability test; (4) reducing the maximum number of context switches for a job execution request by a factor of two" [11]. It thereby addresses dynamic priority scheduling and introduces static preemption levels for shared resources. Stack resources are defined by ν_R denoting the amount of currently available resources and $\upsilon_R(J)$ is the maximal amount job J requires of R. Via calculating system ceiling $\prod_s(t)$ and current static resource ceiling $ceil(p^k)$, a job can derive whether it is allowed to preempt a job or not. For preemption, it is required that the J has the highest priority among the jobs ready to run and that its preemption level is higher than the system ceiling.

6.1.6 DPCP

Dynamic PCP was developed concurrently to the SRP protocol and is another attempt to address dynamic priorities for PCP [16]. It provides partially improved processor utilization compared with EDF, prevents priority inversion, chained blocking, and keeps the system deadlock free. To achieve this, each critical section is assigned a dynamic priority ceiling value, that reflects the earliest deadline of tasks trying to access the resource at a specific time instance. Tasks are blocked if their priority is not higher than all currently used resource ceiling priorities.

6.1.7 MPCP

In [43], Rajkumar presented the first approach to provide mutual exclusive access to resources not only in a multi task environment, but also for multiple processors, where multiple tasks run concurrently (Multiprocessor Priority Ceiling Protocol - MPCP). It is explicitly stated, that only critical section accesses can affect (block) other tasks, since critical section code is very small compared to non-critical execution time. Tasks accessing local semaphores on a processor use PCP locally. Among resources shared globally among processors, global semaphores are used upon a dedicated synchronization processor. Using more synchronization processors that also execute application tasks is denoted as the generalized MPCP. Global ceiling priorities are inherited if a higher priority task wants to access a resource that is hold by lower priority task. In this case, the lower priority task inherits the higher priority task's priority and continues execution. Critical sections can be preempted, if a task wants to access a global resource thats ceiling priority is higher than the currently hold resource and the resources are not the same. With a single dedicated processor for global semaphores, global resource accesses are always executed sequentially.

Similar to DPCP, MPCP extends priority inheritance (priority of a resource holding task is raised to the maximal priority of itself and all tasks waiting for the same resource) with priority boosting via unconditionally elevating a priority temporarily above the highest-possible base priority to expedite the request completion [15].

6.1.8 MSRP

Gai *et al.* extended SRP with preemption thresholds in [27] (SRPT) by letting tasks share pseudo-resources to make them mutually non-preemptive. The idea produces very little overhead, retains advantages of SRP (preventing deadlocks, bounds maximal blocking times, reduction of context switches, supporting multiunit resources), and reduces the maximal stack space requirement by reducing preemption. Therefore, an optimal assignment of preemption thresholds to tasks as well as a set of non-preemptive groups minimizing total stack size is identified. Instead of suspension-based protocols, MSRP uses spinning *i.e.*, busy waiting.

6.1.9 FMLP

The Flexible Multiprocessor Locking Protocol (FMLP) is described in [12]. Initially, instead of directly scheduling tasks to processors, Block *et al.* first link tasks to processors. Consequently, a task can be linked but not scheduled such that the task is non-preemptively blocked or a task is scheduled but not linked such that the task is not one of the highest priority tasks but is scheduled due to it is non-preemptive. This mechanism is denoted as G-EDF (global earliest deadline first) for suspendable and non-preemptable jobs. FMLP further distinguishes between short and long resource holding durations (specified by the user), executes only short requests non-preemptively, and groups resources to efficiently deal with short, non-nestable resources. Such resource groups are protected by queue locks and address nested resource accesses *i.e.*, cases where resource accesses are contained in other resource accesses (*e.g.*, $...lock(r1)$, ..., $lock(r2)$, ... $rel(r2)$, $...rel(r1)$). This mechanism performs more effectively compared with MSRP regarding schedulability.

6.1.10 OMLP

Brandenburg *et al.* introduce in [15] the priority inversion metric (pi-blocking) and present the $O(m)$ locking protocol (OMLP).While considering JLSP (job level static priority) schedulers, Brandenburg *et al.* demonstrate that neither priority inheritance nor priority boosting can be the basis for asymptotically optimal locking protocols. With OMLP, a lower bound on pi-blocking is defined for each resource request that differs whether suspension-aware or suspension-oblivious approaches are addressed. Furthermore, a resource holding job is always scheduled and the duration of pi-blocking is bounded by the maximum request span. By illustrating 'priority donation', it is shown that an 'unlucky' job cannot be preempted repeatedly. "A priority donor is a job that suspends to allow a lower-priority job to complete its request" [14]. Experiments illustrate that OMLP preforms better than all MPCP experiments where spinning (busy waiting) is used and also better than all MPCP results for a resource request probability of 20%. With 40% resource request probability, OMLP only performs better than suspension based MPCP below 50% processor utilization. The main application of OMLP is clustered scheduling.

6.1.11 RNLP

Ward *et al.* [49] compare FMLP and OMLP based on partitioned, clustered and global scheduling as well as spinning, s-aware, and s-oblivious analysis and present the Realtime Nested Locking Protocol (RNLP). Unlike group locks, RNLP provides more fine grained nested resource requests that only requires partial order on resource acquisitions. Therefore, a k-exclusion token lock and a request satisfaction mechanism (RSM) is used. The token lock defines when to react according to incomplete resource requests. It is stated that tasks waiting for a token make progress via priority boosting, inheritance or donation. There are six rules to compete for a shared resource in RSM after receiving a token. For each resource, a queue that is ordered by priority and increasing timestamps defines for "a nested resource request to effectively 'cut in line' to where it would have been had it requested the nested resource at the time of its outermost resource request" [49]. RNLP improves average-case pi-blocking due to fine-grained locking where nested resource requests are not grouped and individual (separate) resources can be hold by jobs concurrently.

6.1.12 PWLP

The Preemptable Waiting Locking Protocol (PWLP) was published by APP4MC project partner colleagues from Regensburg [4] to address disadvantages of the OSEK-PCP for concurrent systems *i.e.*, the possibility of deadlocks to occur and OSEK-PCP's inapplicability to dynamic priority schedulings. With a new mean Normalized Blocking Time metric (mNBT), temporal synchronization effectiveness is evaluated. Based on the resource model from FMLP (Section 6.1.9), PWLP retains advantages of FMLP *i.e.*, supporting partitioned and global scheduling with job-fix, task-fix or dynamic priorities while allowing nesting and further minimizes priority inversion via preemptive busy waiting. Experiments were performed with varying numbers for sharing degree α, resource accesses per task K, periods P, and task utilizations (here 0.1% to 10%) whereas other parameters were static: the number of tasks and processors was set to 1000 resp. 4. With $\alpha = 0.5$ all task sets were schedulable. While FMLP is able to schedule more task sets without deadline violations (in average among all numbers of resource accesses per task) for $\alpha = 4$, $K = 0 \ldots 19$ and periods from 10ms to 100ms, PWLP schedules more task

sets for $\alpha = 1$, $K = 0 \ldots 12$ and task periods of 3ms to 33ms. It is stated that this fact might probably be caused by increased nesting if tasks recur in shorter intervals.

6.2 OSEK-PCP in AUTOSAR

A brief overview of spinlocks used in AUTOSAR was given in Deliverable D2.1 [8], chapter 2.3.6. Furthermore, Gomes *et al.* [29] outline resource sharing problems in AUTOSAR and the corresponding inefficiency of OSEK-PCP like the violation of base priorities, no task suspension during holding a resource causing inefficient and error prone system designs and allocation overheads as well as spinlocks. The fact that tasks waiting for a resource among cores are not ordered further causes more unpredictable blocking times as well as priority inversions to take longer. To address this, Gomes *et al.* [29] propose the concept of dynamic hinting for resource management in order to improve system reactivity. This dynamic hinting is applied to spinlocks that are globally shared resources among cores in AUTOSAR. The idea is that hints are generated upon blocked tasks indicating the lower priority task holding the resource to release it due to its spurious influence. For this purpose, a cross-core waiting priority queue was implemented to provide that always the highest priority task receives access to a spinlock next. Consequently, only the highest priority task will busy wait upon the spinlock while other tasks wait.

The next section briefly introduces a concept to further reduce busy waiting in AUTOSAR based on a priori knowledge about task structures, that are composed of instruction consuming runnables. Therefore, specific time slots of accesses to shared resources within a task are identified and based on runnable causality and task release deltas (system state), runnable orders are shifted to reduce the total amount of resource conflicts.

6.3 Resource Management Concept for APP4MC

Instead of introducing a new protocol and with the knowledge about runnable orders and label accesses, we can estimate shared resource access *collisions* depending on conflicting task release time (RT) and delta values $\mathrm{RT}_{t_x} - \mathrm{RT}_{t_y} = \delta_{t_x \to t_y}$. The approach is denoted as Task Delta based Runnable Rescheduling ($TDRR$) that defines Runnable Schedule Orders ($\mathcal{RSO}(c, \Delta_c)$ that depends on a conflict c and the task release delta values within Δ_c that defines multiple $\delta_{t_x \to t_y}$ values for different tasks t_x and t_y.

The following Figure 6.1 illustrates an example case showing three tasks, consisting of three runnables each, whereas the first task accesses a lock at its first runnable, the second task at its second runnable, and the third task at its last runnable.

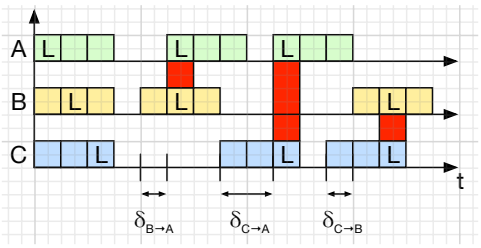

Figure 6.1: Three conflicting tasks, 3 label accesses

Within the $\mathcal{RSO}$ calculation, different task δ values are considered. Since a task cannot have a delta release times to itself, the matrix's main diagonal is 0 if there is no conflict in case all tasks are released at the same time or 1 if there is a conflict for the same release time across all conflicting tasks. Another example is given in Figure 6.2.

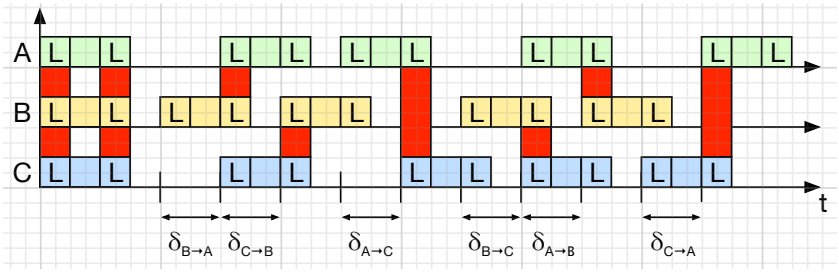

Figure 6.2: Three conflicting tasks, six label accesses, conflicts considered between two tasks

The actual release times of tasks vary and depend on the system state. If the Δ matrix has multiple values in a column > 0, then a $\mathcal{RSO}$ must be calculated for the current task while considering multiple tasks of the corresponding column.

Busy waiting periods occur when a runnable's request to a lock is rejected since another runnable is already holding the lock (cf. Section 6.1, Figure 6.1, red marks, or Figure 6.4, case (e)).

Runnable schedule orders $\mathcal{RSO}$ can be calculated that define for each conflict c and potential task release delta values that would result in busy waiting, a schedule order for a latter released task. In particular, at each release of a task, the system is observed whether one or more conflicting tasks are already executing (i.e. they were release earlier). If this is the case, the task is released with a runnable schedule order that minimizes busy waiting.

Consequently, as already stated in the task declaration, a $\mathcal{RSO}$ is a task instance i.e. a concrete permutations of a ProcessPrototye. Such permutation can differ from permutations of other instances of t_m that already finished their execution.The concrete permutations (runnable schedule orders) are calculated to minimize busy waiting periods ($\mathcal{BW}(c, \Delta_c)$) as described in the following along with an example.

Having static runnable orders, busy waiting can appear at any given instance of time and causes unpredictable delayed task release times, hence, less deterministic behavior. In order to illustrate the described problem and its corresponding solution, the following directed acyclic graph (Figure 6.3) is given, consisting of runnables $r_0 - r_9$, that each consume a single instruction (time instance) for simplification and comprehension purposes. Runnables have directed dependencies (edges) denoted with arrows.

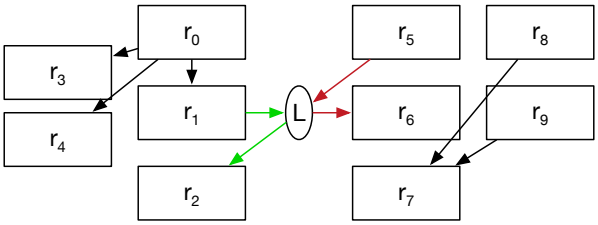

Figure 6.3: Runnable dependency graph with two tasks and a shared label

The two tasks t_0, t_1 share one common label L, that is written by r_1 and r_5 and read by r_2 and r_6. Due to concurrent system architecture, the label must be protected by a lock *e.g.*, a semaphore. Assuming the spinlocks for globally shared resources in AUTOSAR, the task is granted access with the highest priority causing the lower priority task to wait until the resource's lock is released (spinlocks address the tasks being assigned to different hardware processing units resulting in concurrently progressing tasks). Using the $\mathcal{RSO}$, such busy waiting can be reduced to keep response times low and retain deterministic behavior. Possible execution results are illustrated in Figure 6.4.

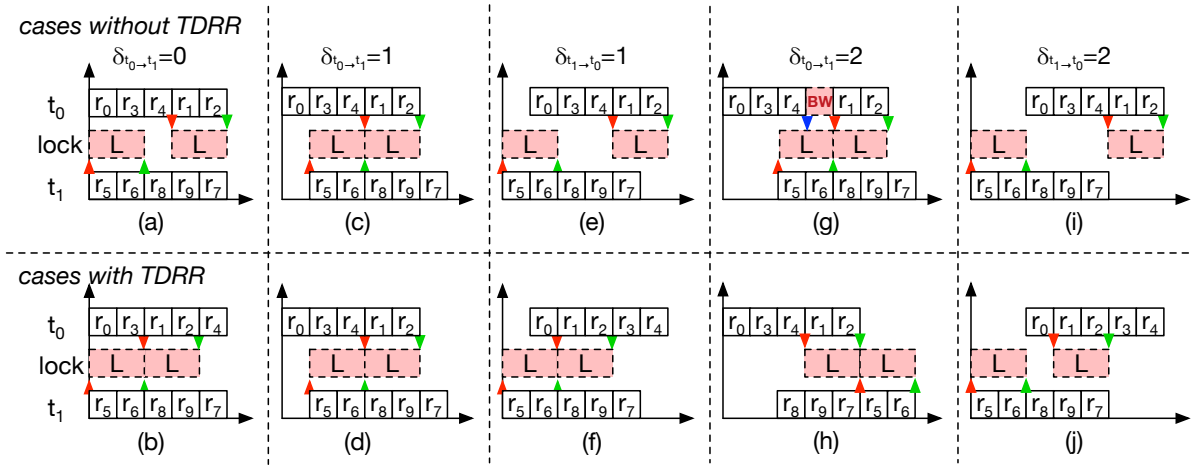

Figure 6.4: Timline diagrams showing the execution of tasks of Figure 6.3 regarding five different task release deltas with and without $\mathcal{RSO}$ utilization

Figure 6.4 shows ten possible concurrent executions cases $(a) - (j)$ of the two tasks (t_0, t_1) with five runnables each, in regard to five different release deltas (columns): $\delta_{c_0 t_0 \to t_1} = \{0, 1, 2\}, \delta_{c_0 t_1 \to t_0} = \{1, 2\}$. Furthermore, cases for $\delta_{c_0 t_0 \to t_1} = \{3, 4\}, \delta_{c_0 t_1 \to t_0} = \{3, 4\}$ are shown in Figure 6.5.

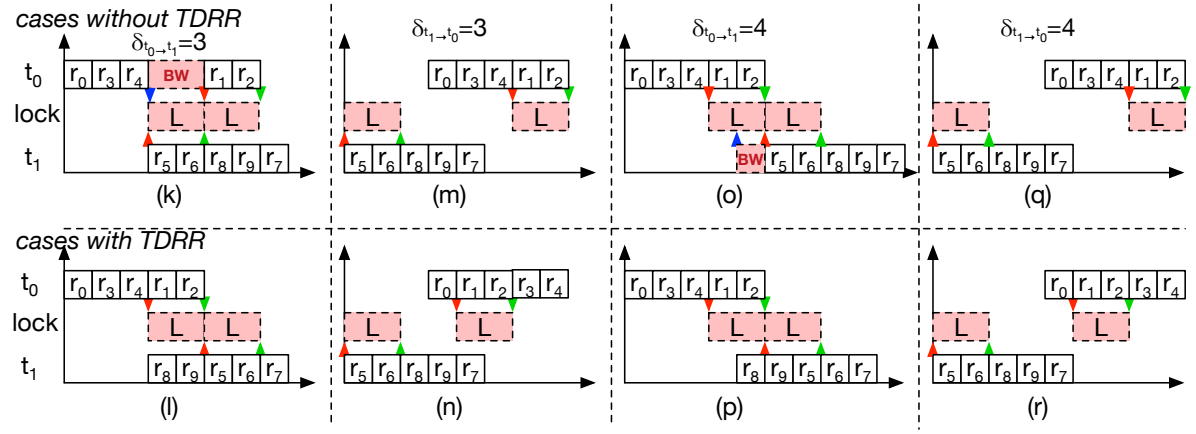

Figure 6.5: Timline diagrams showing the execution of tasks of Figure 6.3 regarding further four task release deltas with and without $\mathcal{RSO}$ utilization

Accesses to the shared label respectively its lock are denoted as red arrows (lock granted), green arrows (lock released), and blue arrows (rejected accesses). Such rejected accesses caused by a locked resources result in blocked tasks that will busy wait until the locks are released and available. The task released first is assumed to the reveal static runnable schedule order. The latter released task uses the pre-calculated $\mathcal{RSO}$ if it predicts busy waiting based on the task release delta Δ_c *e.g.*, cases (g), (k), and (o) are replaces with (h),(l), and (p) such that busy waiting is reduced completely.

In case multiple tasks are waiting for the same resource, the concept from [4] could be used in order to assign the resource to tasks with highest priority and longest polling periods first. Without TDRR (first rows, cases (a),(c),(e),(g),(i),(k),(m),(o),(q)), the runnable execution order remains static and may be blocked by a locked resource as in case (g), (k), and (o). With TDRR, (second rows: cases (b),(d),(f),(h),(j),(l),(n),(p),(r)), the runnable execution order is based on the pre-calculated $\mathcal{RSO}$ and no busy waiting occurs in case (h), (l), and (p), resulting in a reduced response time. In this example, only three schedule orders for case (h),(l), and (p) are be calculated. It may be of interest, to keep lock requests close to each other, shown in other cases with TDRR. Such narrowed down locking time may grand the resource to further tasks and reduce busy waiting there. These calculations are intended to be configurable, such that either the static order from the first row is executed or the $\mathcal{RSO}$ are calculated and executed correspondingly. To sum up, the amount of busy waiting reduction consequently depends (1) on the amount of conflicts $\mathcal{C}$, (2) the runnable dependency structure that is derived from the graph $\mathcal{DAG}$ (the $\mathcal{DAG}$ also defines the conflicting runnable's shifting times ts_{r_r}, ts_{w_r}), and (3) on the task release delta Δ_c.

$\mathcal{RSO}$ are selected at release time and will be static for a single instantiation of the task. TDRR intends to provide $\mathcal{RSO}$ for as many different system states as possible. However, the amount of cases that have to be investigated for busy waiting rises significantly very early as shown in the following example.

A simple fictional system consists of task t_0 sharing a resource A with tasks t_1 (c_0) and another resource B with task and t_2 and t_3 (c_1). t_0's instruction length is 3 and $I_{t_1} = 100, I_{t_2} = 50, I_{t_3} = 10$ correspondingly. The overall number of task delta combinations for the tasks is already 680. It has further to be investigated, if $\mathcal{RSO}$ should be calculated offline for potential conflicts and how the amount of delta combinations can be effectively reduced.

6.4 Summary

The previously described TDRR approach conceptually seems promising for reducing busy waiting that occurs in multi processor systems when shared resources are managed via spinlocks *e.g.*, in AUTOSAR. Various simulations and evaluations need to be performed in order to validate the benefits of TDRR. Furthermore, the solution space exploration requires specific restriction techniques since multiple shared resources can be accessed by multiple runnables along with various task release deltas while each combination reveals different runnable order combinations that have to be investigated. Since schedule tables also affect other label accesses, a good idea might by to initially abstract from runnable dependency graphs to resource graphs (*e.g.*, shown in [11]). This would identify which tasks influence other tasks in case their runnables are reordered and which tasks can be analyzed independently. The graph from Figure 6.3 would result in simply $t_0 - t_1$ (undirected, denoted inter task communication graph

(ITCG)), denoting that t_0 influences t_1 and the other way around. Such shared resource graphs would have to be taken into account for calculating the $\mathcal{RSO}(c, \Delta_c)$.

It has to be shown that TDRR provides a more deterministic behavior and less busy waiting compared to AUTOSAR or currently available approaches. It can further be assessed regarding the protocols presented in section 6.1. The proposed TDRR concept is intended to be implemented and tested along with APP4MC as well as evaluated with simulation tools (e.g. Timing Architects Toolsuite) in the near future. Also, hardware execution combined with tracing is intended to be used to validate the results.

7 Case Study

7.1 Parallax ActivityBot

The Parallax ActivityBot has been briefly described in AMALTHEA4PUBLIC deliverable D2.1 [8]. The following sections provide a more detailed insight into the progressed use of the ActrivityBot.

7.1.1 Application Description

The application described in the following was used along with the summer school 2016 at Dortmund University of Applied Sciences and Arts. It includes a simple structure that is modeled in APP4MC to utilize parallelism possibilities of the ActivityBot. The application is intended to do the following:

- Click on a button
  - robot drives until an obstacle in front is detected and then turns to the **right** by a $\sim$ 90 degree angle
  - at the same time , two LEDs are blinking: LED 26 and LED 27 in different frequencies (LED 26 $\rightarrow$ 5Hz, LED 27 $\rightarrow$ 2,5Hz)
- Another click on the button
  - robot continues to drive until obstacle in front is detected and then turns to the **left** by a $\sim$ 90 degree angle
  - the two LEDs 27 and 26 are blinking in swapped frequencies (LED 26 $\rightarrow$ 2,5Hz, LED 27 $\rightarrow$ 5Hz)

7.1.2 Modeling

The application was structured into the following 23 runnables:

Activation	**Runnables**
1000ms	pin11, buttonPushed, pause(1000)
200ms	LED26(200ms), LED27(200ms)
125ms	init, servoLeft, servoRight, driveForwardsButtonStateTwo, driveForwardsButtonStateOne, buttonStateOne, buttonStateTwo, driveLeft, driveRight, click
100ms	LED26(100ms), LED27(100ms)
220ms	ping, distance, pause(200ms)
12ms	abdrive, setRampStep, rampStep

Table 7.1: Runnable's activation references

The runnables were traced to derive the following instructions:

Runnable_Name	Instructions	Runnable_Name	Instructions
distance	97008	pause(200)	2400000
driveForward-ButtonStateOne	228048	buttonPushed	8292
driveLeft	227904	pin11	8544
driveForward-ButtonStateTwo	228048	pause(1000ms)	12000000
init	8496	buttonStateTwo	228048
servoRight	8340	driveRight	228049
ping	97008	servoLeft	8352
rampStep	120684	abdrive	8880
LED26(100ms)	1200012	click	8544
LED27(100ms)	1200103	setRampStep	8880
		LED26(200ms)	2400042
		LED27(200ms)	2400096

Table 7.2: Instruction Constants of Runnables

Dependencies (`RunnableSequencingConstraints`) are shown in Figure 7.1.

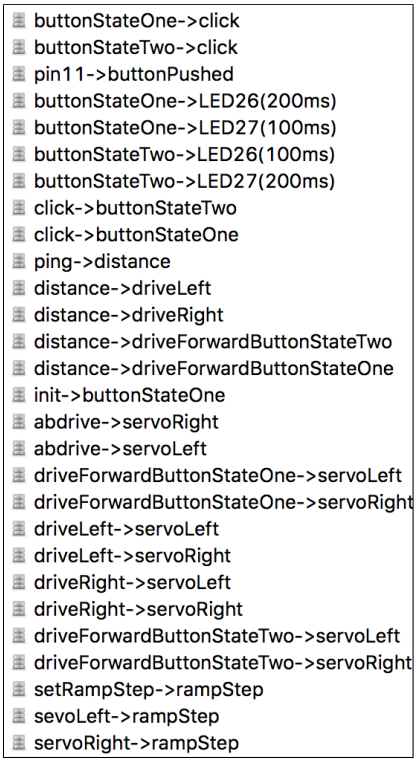

Figure 7.1: ActivityBot software model

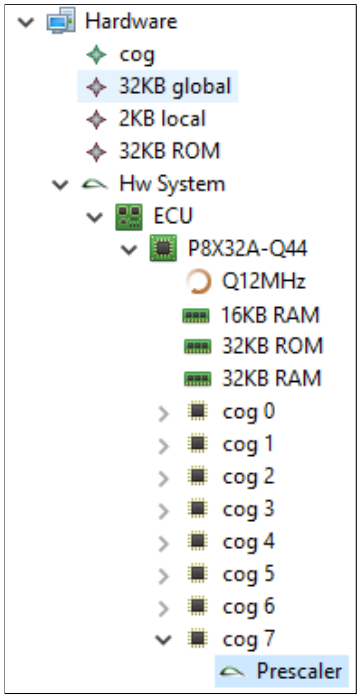

Figure 7.2: ActivityBot hardware model

Additionally, the partitioning must be constrained to prevent separating specific runnables. This separation prevention can be addressed by the runnable pairing constraints, contained

in the affinity constraints. The following three pairing constraints were created to bind the mentioned runnables and ensure a correct execution of the program:

1. pin11, buttonPushed, pause(1000ms)
2. ping, distance, pause(200ms)
3. abdrive, setRampStep, rampStep

These runnable pairings are considered as described in Section 2.2.4. Furthermore, a hardware model was created as shown in Figure 7.2. It contains the following memories:

- name: 32KB global, size: 32786, type: RAM
- name: 2KB local, size: 2048, type: CACHE
- name: 32KB ROM, size: 32786, type: FLASH_INT

Additionally, the elements for the ECU, the micro-controller, the 8 cores, a prescaler, and all corresponding properties were defined. Based on this model, the mapping methodology described in chapter 3 could be utilized.

7.1.3 Code Generation

The application's basis was implemented within a Yakindu state chart model, that was further used to generate code. The state chart model is shown in Figure 7.3.

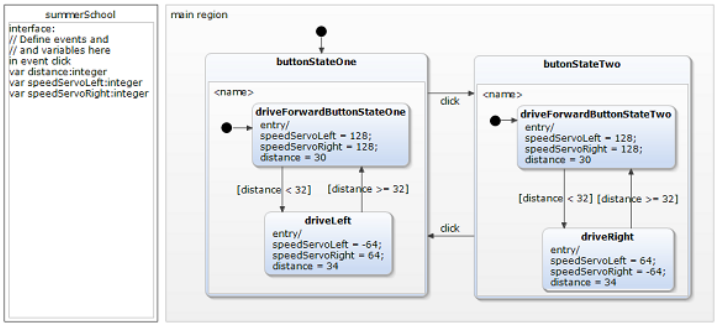

Figure 7.3: ActivityBot state chart model

The generated code required certain adaptations and extension to be finally compiled to the ActivityBot using the SimpleIDE tool. Such adaptation especially address triggering the servo motors, accessing the ActivityBot's buttons in order to change between the button states, reading the ultrasonic sensors, and the LED setup required to set the specific pins that are related to the LEDs.

Partitioning and mapping results were implemented via `cogstart`, `cog_run`, and `cog_end` commands that define which functions run on which cores.

The result was a working parallel application that provides an efficient distribution of software functions to the different cores of the ActivityBot. The presented steps become necessary for bigger projects especially in the automotive domain. The development highly benefits from automatic parallelization in form of partitioning and mapping as well as a common and standardized data exchange model.

7.2 RC-Car XMOS

The RC-Car application has been briefly described in APP4MC deliverable D2.1 [8]. The following provides a more detailed insight into the progressed use of the RC-Car.

The most important progress is the addition of a raspberry pi 3 (RPI3) to the platform. With four cores each of the speed 1.2GHz, it (1) has powerful processing cores required for image processing, (2) consumes relatively low energy (5 Volts, 1-2 Ampere → 5-10 Watts), (3) is impressively cheap, and (4) provides existing interfaces and implementations regarding Ethernet data transmission, OpenCV library usage and camera drivers.

The new RC-Car's architecture is shown in Figure 7.4.

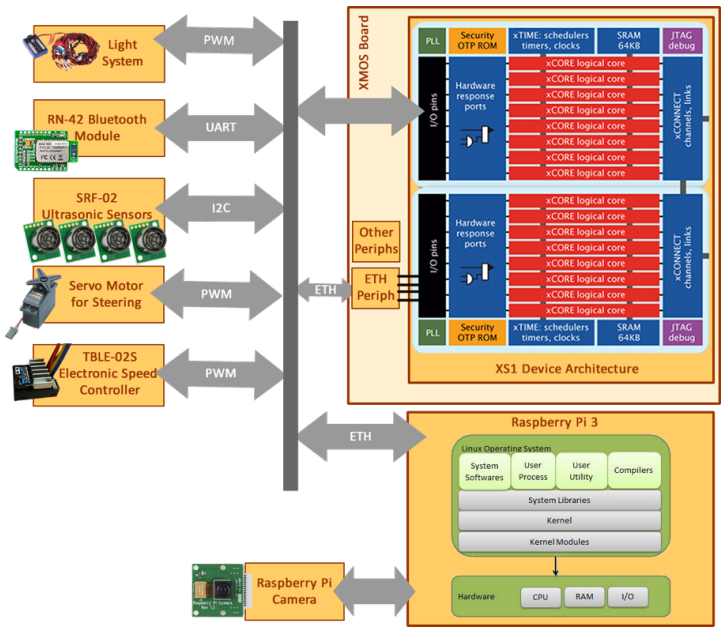

Figure 7.4: RC-Car's architecture

The remote control Android application has been extended to also support a Joystick GUI. In addition, the RC-Car application can perform basic traffic cone detection and corresponding steering with the help of the PRI3 and its Ethernet interface connected to the XMOS ExplorerKit board.

The AMALTHEA model for the RC-Car application is still under development. We intend to use *property constraints* for binding the OpenCV implementations to the RPI3 and *affinity constraints* for the partitioning to prevent mixing up OpenCV and steering respectively RC-Car related functions.

In the later course of AMALTHEA4PUBLIC, we plan to investigate and evaluate the parallelization features partitioning and mapping for the RC-Car with APP4MC. This plan is also intended to be subject to a related publication.

7.3 Multicore Communications API Implementation

The concepts of the Multicore Communications API (MCAPI), developed by the Multicore Association (MCA), have been briefly introduced within deliverable D2.1 [8]. The next step was

to implement selected parts of this application programming interface for the NXP MPC5668G[1] multicore processor for demonstrating these concepts within the range of AMALTHEA4PUBLIC. This chapter first describes the basic concept of this implementation and the design of the different modules. Moreover, a test scenario which shows a communication between the two cores of the MPC5668G, using the MCAPI, will be described.

7.3.1 Implementation Concept

For best portability and reusability the implementation uses a layered architecture. On the top layer the MCAPI functions, as defined in [47], are located, which also include the logic and semantics for error handling. Under the MCAPI layer lies the transport layer, whose functions will be initiated by the MCAPI functions. The functions of this layer are responsible for creating a MCAPI environment, the actual transfer of data between the two cores and finalizing the MCPAI environment when all communication is done. Through a strictly separation of these layers there will not be any cycles inside the dependency graph because only functions from a higher layer have access to functions below and not the other way around. Underneath the transport layer are functions that are used for message queues within the data transfer and functions with direct access to the hardware of the MPC55668G (e.g. acquiring a hardware semaphore). So changing to a different target platform is possible without much effort. Only the hardware dependent parts have to be replaced and the layers above remain unaffected. The following section will give a deeper insight into the individual modules.

7.3.2 Module Design

The module design is based on a stub implementation done by the MCA board member NXP Semiconductor.

Transport Layer

The transport layer contains all functions for transferring data using the messages concept of the MCAPI. This includes for example creating a node, encoding and decoding a message transfer handle and getting the IDs of a node. The most relevant functions will be describe below.

- **mcapi_trans_initialize():**
  With this function a MCAPI environment will be created for a calling node. If this function is called the first time, the database address is defined and the node will be added to the database with its domain ID and node ID. Trying to initialize a node with IDs already used or adding a node where there is no empty space in the database will result in an error.

- **mcapi_trans_endpoint_create():**
  Each initialized node needs at least two endpoints for communication. One for sending data and one for receiving data. Calling this function will create an endpoint which could be identified by the parameters domain ID, node ID and port ID. After checking if the maximal number of endpoints has not been reached, the endpoint will be added to the

[1] `http://www.nxp.com/mpc5668G`, online, access June 2016

database. Afterwards a handle will be created using the three IDs to identify the endpoint in the system.

- **mcapi_trans_send_data():**
  This function is responsible for transferring the data via a buffer. First of all the queue of the receiving endpoint is checked if it is already full. If the checking function returns positive, the next empty entry will be searched and the index will be returned. Data will be copied into a buffer and the buffer will be linked to the queue entry.

- **mcapi_trans_recv_data():**
  For receiving data, the corresponding entry will be removed form the queue and the index of the buffer will be stored temporary. After checking if the application buffer has at least the same size as the transfer buffer, the data will be copied into it. Once the receiving process is complete, the transfer buffer could be used for another communication.

- **mcapi_trans_finalize():**
  At the end of all communications, the nodes have to be finalized before closing the MCAPI environment. Thus every node has to call **mcapi_trans_finalize()** whereby this node will be marked as invalid. If the calling node is the last node in the system, the MCAPI environment will be closed.

A closer look at these functions and their corresponding sequence diagrams could be found in the following section 7.3.3.

Database

The MCAPI specification defines a precise topology of an MCAPI environment. At the top of every system a domain or multiple domains are located. Each domain could have one to many nodes (implementation specific) whereby each node could create one to many endpoints (implementation specific). The topology created for this implementation is shown in figure 7.5. The database elements are realized as struct data types. At the top of this implementation is the *mcapi_database* struct. It contains the domains created in an environment, the buffers for transferring the data and a counting variable for the number of domains in the system. The *trans_buffer* struct contains a variable for transferring scalar values (*scalar*) using the scalar channels, which could be implemented in a further version. If a buffer is already in use, it will be marked using the *selected* variable. The actual size of the data will be stored in *size* and the data itself in *buff*. For this example ten buffers could be used.

Each domain in a system (two for this example) has its unique *domain_id* with which it could be identified. This ID will be set during initialization of the first node in the specific domain. The variable *valid* signals if a domain is still valid and is important for finalizing the whole system. Moreover, the domain struct contains the created nodes and a counter for the number of nodes inside a domain.

For identifying a node, a *node_id* has to be set during initialization. This ID could not be changed during runtime. In this example the maximum number of nodes is five. Marking a node as valid is similar to the domains, using the *valid* variable. With *endpoint_count*, the actual number of endpoints related to a node could be queried.

An endpoint (up to ten in this example) is indicated by its *port_id*. Similar to domains and nodes *valid* provides information about the validity of an endpoint. For a subsequent extension of this implementation, the endpoint struct contains variables for checking if an

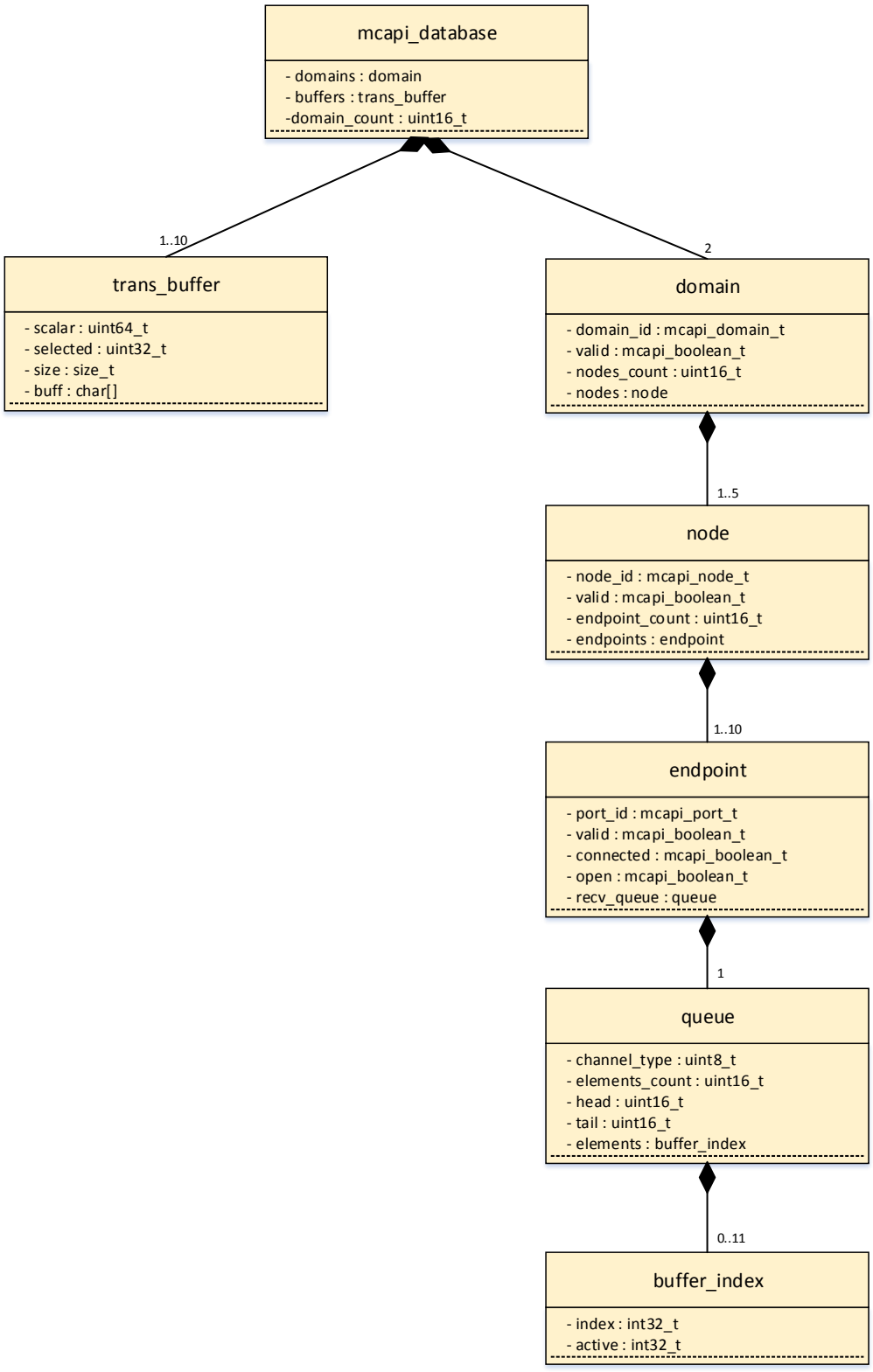

Figure 7.5: Datbase model for example implementation

endpoint is connected to a channel (*connected*) and if connected, the channel has been opened for transferring data (*open*). To process the messages in order they were send, a reference to the buffer will be added to the receive endpoint *recv_ queue*. Each endpoint has one receive queue.

In the receive *queue* struct following information are stored: *channel_ type* (defines whether data is transferred via channels), *elements_ count* (number of elements in the queue), *head* and *tail* (indicates the first and last element in the queue) and a *buffer_ index* component to link a buffer to a message communication. For this functionality the *buffer_ index* component contains an *index* and an *active* variable which store the buffer index and signals whether the buffer is in use. In this implementation a queue consists of a maximum of eleven elements.

Queue and hardware layer functions

Transferring the data is done by using buffers which are referenced in the receive endpoints queue. The receive queue is realized as a circular buffer with an implementation specific size. If the queue is full, no further elements could be added to the queue until the application on receiver side removes elements. The queue component contains four functions for controlling the queue. With the function *push_ queue()* an element is added to the queue and the index is returned to the calling function, indicating the position of the added element. Moreover, the counter variable and the pointer to the last element will be incremented by one. For getting an element from the queue, the function *pop_ queue()* is used. It returns the index belonging to the buffer with the data. The counter variable will be decremented by one and the pointer to the first element will incremented by one. Due to the fact that the queue operates on the FIFO principle, every time *pop_ queue()* is called, the oldest at that time unread element is taken from the queue.

Additionally to the pop and push functions, the queue module contains two functions for checking whether the queue is full or empty. *full_ queue()* checks if the queue is full by comparing the actual number of elements in the queue with the maximum specified in the requirements. Thus it is not possible to remove an element from an empty queue, *empty_ queue()* checks if there are any elements in the queue by comparing the counter with 0.

For this implementation the hardware semaphores of the MPC5668G were used. These functions were defined in the module *semaphore_ mpc()*, which only contains functions with direct access to the hardware registers of the MPC5668G. This enables a good portability to another target platform, since only this module has to be implemented for a new processor.

In order to prevent conflicts when accessing the database, the semaphores are used. Before an access occurs, a semaphore will be locked, so that only one core of the MPC5668G could operate on the database. To identify which core uses the semaphore the processor ID of each core will be used. In the special-function register (SPR) 286 the ID is stored. With a single instruction the ID could be moved to a general-purpose register for further processing. With this ID the function *gateLock()* could lock a semaphore for the calling core by writing a lock value (processor ID + 1) to the semaphore register. If the other core tries to lock the semaphore, it has to wait in a spin-wait loop, until the semaphore is released by the first core. Releasing a semaphore is done by calling the function *gateRelease()*. In this functions the semaphore is released by writing 0 to the register.

7.3.3 Communication Example

For illustrating a message communication via the MCAPI, this section shows an example communication using the MPC5668G. Therefore two domains with each containing one node will be created. One domain belongs to one core of MPC5668G. For sending and receiving data, two endpoints will be created per node. The sequence of each step will be shown in a sequence diagram.

Initializing

The first step in every communication process is initializing the MCAPI environment by creating nodes. Each core gets its own domain ID and node ID. In the main-routine of the respective core the process of creating a node is started by calling *mcapi_initialize()* (shown in figure 7.6). This function may be called once per node and a node has to be finalized before calling *mcapi_initialize()* again. On the MCAPI layer error handling is done by analyzing the results of the transport layers checking functions. With *mcapi_trans_valid_node()* and *mcapi_trans_initialized()* will be reviewed whether the node parameters are valid and whether the node has already been initialized. Within the checking function *mcapi_trans_initialized()* the function *mcapi_trans_get_ids()* returns the IDs of the calling node for comparing these with the actual parameters. *Processor_ID()* is used to identify which core is calling the initialized function.

After completing the checking functions, by calling *mcapi_trans_initialize()* the initialization will be started. In a first step the processor ID will be requested and the database will be locked by calling *gateLock()*. In the next step the database will be initialized. Afterwards some checks will be done, ensuring that there is no domain with the same ID already contained in the database and whether the maximum number of nodes is reached for this domain. Only if all tests are successful, the node will be added to the database with its specific IDs. In the end, the semaphore will be unlocked and returning *MCAPI_TRUE* indicates that the initialization was successful.

After initializing the nodes, the next step is creating one or many endpoints depending on the requirements for a communication.

Creating an endpoint

By calling the function *mcapi_endpoint_create()*, the process of creating an endpoint will be started, which could be seen in figure 7.7. At first the domain ID will be returned by calling *mcapi_domain_id_get()* and the transport layer function *mcapi_trans_get_doamin_num()*. With subsequent checks it will be proved, whether an endpoint with the port ID already exists (*mcapi_trans_endpoint_exists()*), the maximum number of endpoints for a node is reached (comparison of maximum value with result of *mcapi_trans_num_endpoints()*) and whether the port ID is in accordance with the specifications (*mcapi_trans_valid_port()*). Ends none of the checks in an error, the endpoint could be created by calling *mcapi_trans_endpoint_create()*. Since calling this function leads to a database access, the database must be protected from simultaneous access by locking a semaphore. In the next step the IDs will be requested and the next empty index for a new endpoint will be searched. The endpoint will be added to the database and a handle will be generated to identify the endpoint in the environment. A handle consists of the domain index, the node index and the endpoint index combined with a shift value. After completing the creation the semaphore will be unlocked and *MCAPI_TRUE*

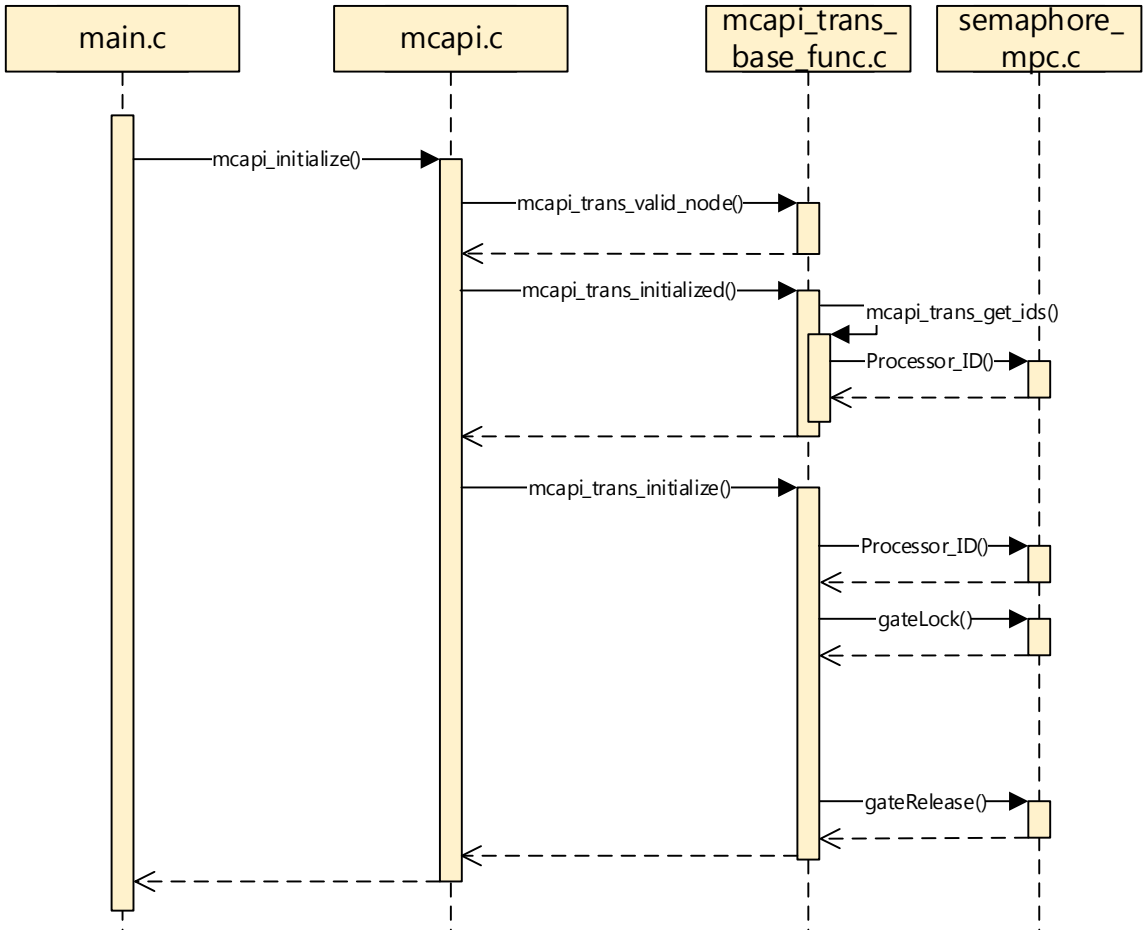

Figure 7.6: Sequence diagram for initializing a node

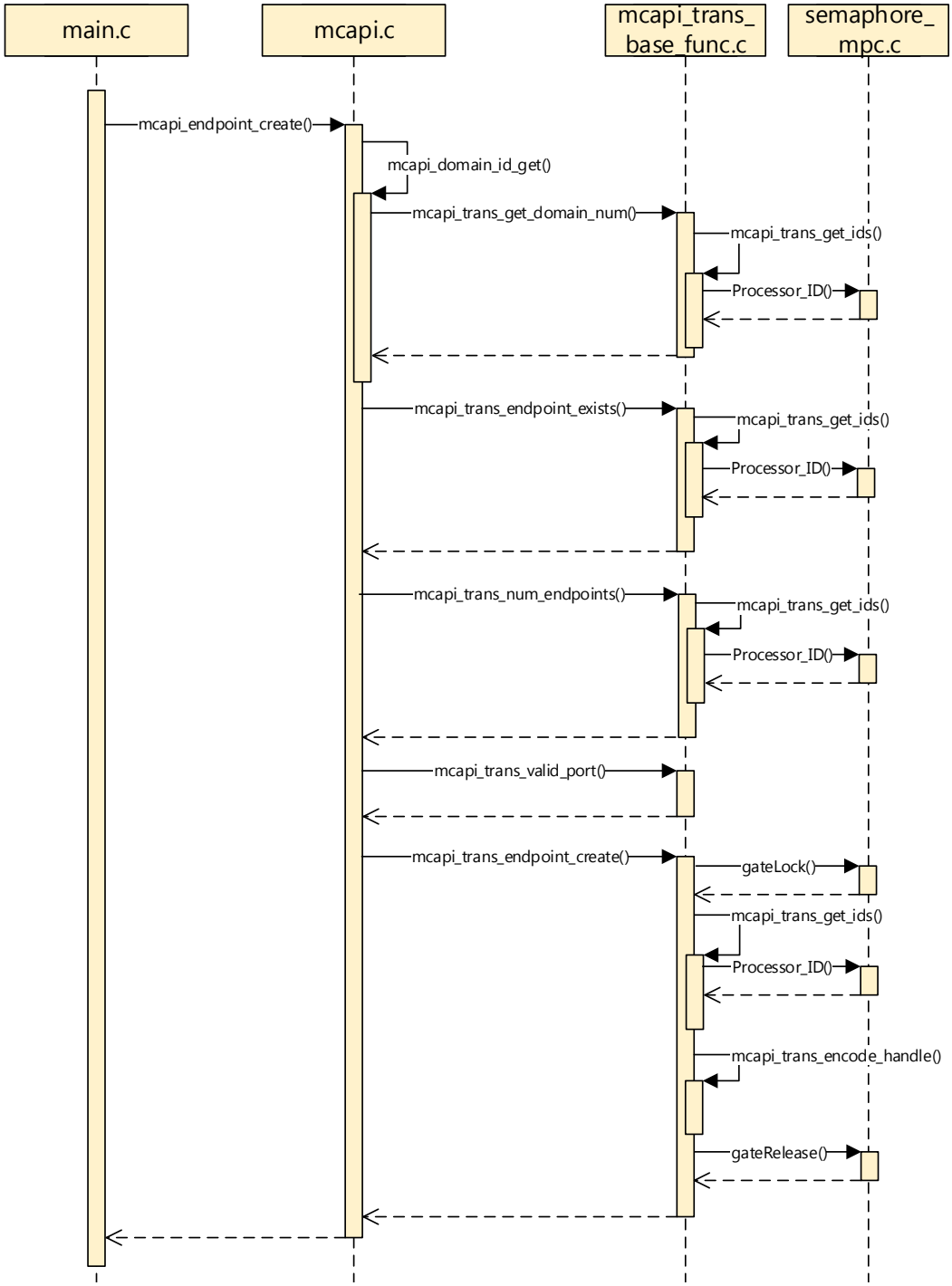

Figure 7.7: Sequence diagram for creating an endpoint

indicates a successful process. With return the created handle, the endpoint could be used sending and receiving data from the main-routine. The next step will be sending data from one endpoint to another.

Sending data

The sending process (seen in figure 7.8) starts by calling *mcapi_msg_send()*. The function is

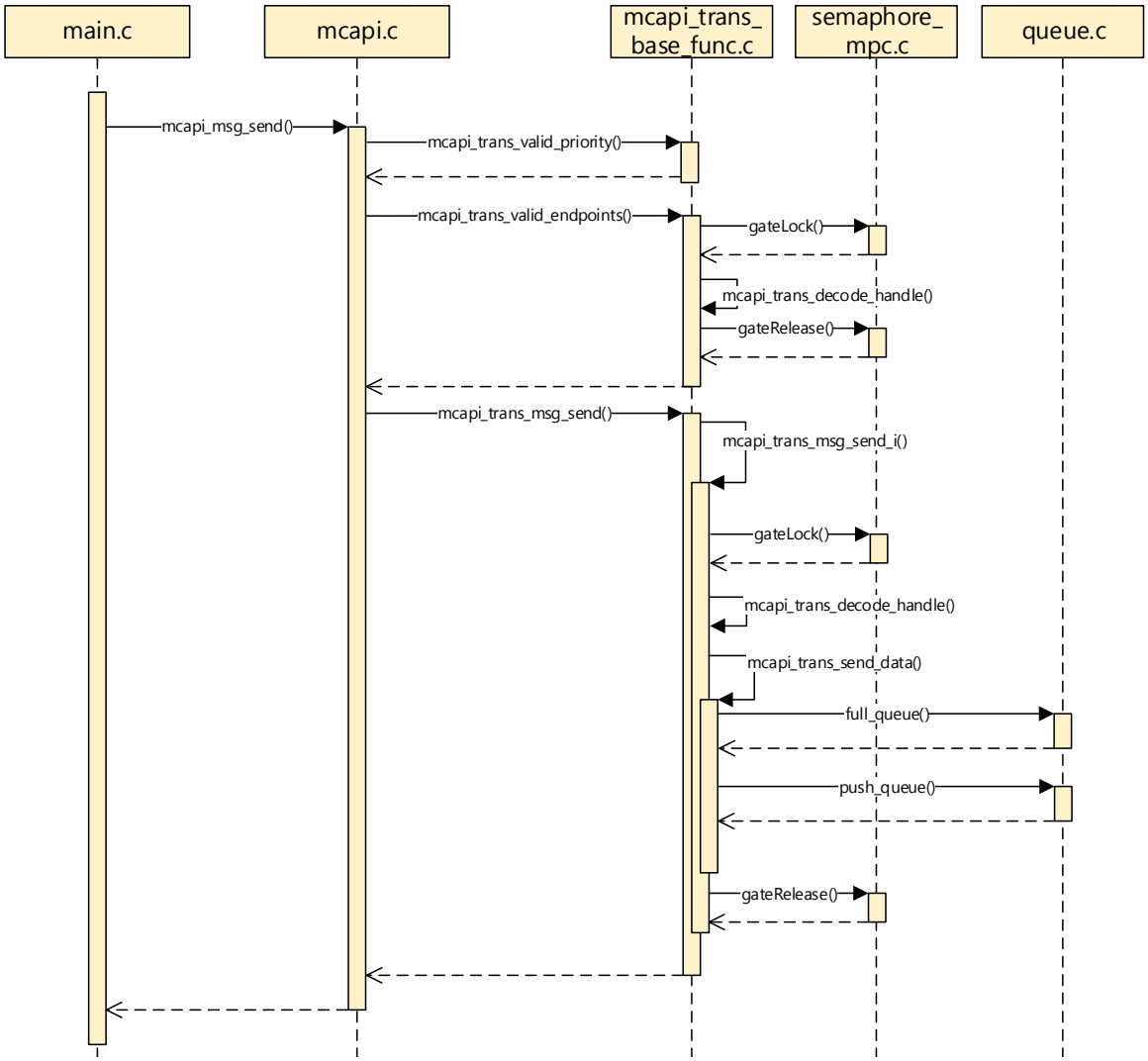

Figure 7.8: Sequence diagram for sending data via messages

called with the parameters start and destination endpoint, the data which should be transferred, the size of the data, a possible priority for each message and a status code which indicates success or failure of a sending process. Before sending the data it will be checked, if the priority is valid (*mcapi_trans_valid_priority()*) and in the next step whether the endpoints are valid (*mcapi_trans_valid_endpoints()*). Inside the function, the database access for another node will be locked and the endpoint handle will be decoded in the indices. In order to ensure that the message size is not bigger than the buffer size, a comparison of the values will be done on

the MCAPI layer.

The actual sending process is initiated by calling *mcapi_trans_msg_send()* and inside this function *mcapi_trans_msg_send_i()* (_i stands for immediately). Splitting these functions will serve as a basis for prospective functions, which implement blocking sending. A semaphore will be locked and the receiving endpoints handle will be decoded in its indices by using the *mcapi_trans_decode_handle()* function. With the *mcapi_trans_send_data()* function, the sending will be realized. This function is also provided for sending data via scalar and packet channels. Before sending, the queue will be checked whether it is full or not. If there is an empty entry, a free buffer for sending data will be searched, placed into the queue by calling *push_queue()* and marked as active. Afterwards, the data will be copied into the buffer and returning *MCAPI_TRUE* indicates a successful sending process. The semaphore will be unlocked and the status is set to *MCAPI_SUCCESS*. The data remains in the buffer until the application, which owns the receiving endpoint, calls *mcapi_msg_recv()* to get the data.

Receiving data

Receiving data works similar to sending data. The process is initiated by calling the function *mcapi_msg_recv()* and after checking if the applications receive buffer corresponds to the specifications (*mcapi_trans_valid_buffer_param()*) and whether the receive endpoint is valid (*mcapi_trans_valid_endpoint()*), the actual receiving process will be started by the (*mcapi_trans_msg_recv()*) function. It follows the same procedure as sending the data, with the difference, that an element is taken from the queue (*pop_queue()*) and the data is copied into the applications receive buffer. Moreover, the transfer buffer will be released and is ready for reuse and the semaphore will be unlocked. *MCAPI_TRUE* will be returned, indicating a successful receiving process.

In this example for the MPC5668G, the received data will be processed and send back to the other core by using two additional endpoints for sending and receiving. After finishing the communication, the nodes will be finalized by calling *mcapi_finalize()* and the MCAPI environment will be closed.

To sum up, the MCAPI environment and the implemented functions provide a lightweight interface for transferring data between nodes. The functionality is kept simple and fast for use on embedded systems.

7.4 Linux as Amalthea-target Platform

A further planned demonstrator should be realized using a linux with real-time capabilities as a target platform running on raspberry pi or beagleboard. Due to the fact that many developers already own one of these platforms, they could easily start using the Amalthea Tool Platform. Linux itself is not real-time capable. But there are several, more or less difficult to implement, approaches which turn linux into a classic real-time operating system, like using a second smaller kernel next to the linux kernel which provides the hardware infrastructure to the real-time tasks (using a real-time hardware abstraction layer (RTHAL)) and manages the interrupts. An example for this approach is RTAI[2] (Real Time Application Interface). Using

[2]https://www.rtai.org

RTAI, the linux kernel itself will be treated as a real-time task with the lowest priority (idle task) and only runs when no real-time task with a higher priority is currently running. The real-time tasks communicate with each other using mailboxes and semaphores while communicating with non real-time task occurs via shared memory and FIFOs. [38]

Due to its structure, implementing real-time applications using the RTAI needs much programming effort. Applications have to be divided into real-time and non real-time parts and a communication path between these parts must also be developed. In order to achieve hard real-time requirements, applications must be executed in kernel space, where fatal errors can lead to a crash of the entire system. But nevertheless RTAI has become a stable solution for making linux real-time capable which is still maintained with the latest test version released in May 2016.

Another, more popular, approach turning linux into a classic real-time operating system, is using the PREEMPT_RT patch, mainly maintained by Ingo Molnár and Thomas Gleixner. During the last years, most of the PREEMPT_RT functions were merged to the mainline linux kernel, with only a few functions left which have to patched to make the kernel full preemptable. Moreover, the Real Time Linux collaborative project was established by the Linux Foundation[3], to support the developers and provide the needed resources for mainlining the PREEMPT_RT. With using the PREEMPT_RT patch there is no need of a second API. All function are included and the programmer can use the POSIX programming interface. New locking and synchronization functions are also included. To make the locking primitives preemptable, spinlocks were replaced by rtmutexes with those higher prioritized tasks could preempt tasks with a lower priority. This solves the priority inversion problem. Real-time tasks could run in user-space where debugging is much easier. Moreover, real-time tasks could get a priority from 1 to 99, with 99 representing the highest priority. These real-time tasks get as much computation time as they need. Normal tasks get computation time when all real-time tasks have finished their work. To set a priority, POSIX provides the function *sched_setparam()*. In a multicore system you need to assign tasks to different processors, which could be done by calling *sched_setaffinity()*. Furthermore PREEMPT_RT includes high resolution timer (hrtimer) with a resolution in nanosecond range. Real Time Linux supports the most common hardware architectures like the Power architecture, ARM and x86. [23], [38]

Summing up, real-time with linux is possible more than ever and so it could be a good opportunity using it for a Amalthea4public demonstrator. With RTAI and the PREEMPT_RT patch linux is capable for real-time applications with the PREEMPT_RT patch beeing the popular approach particularly because it is mainlined in the linux kernel and supported by the Linux Foundation.

[3] http://www.linuxfoundation.org/collaborate/workgroups/real-time

8 Conclusion

In Chapter 2 we investigated so-called *list scheduling* and *bin packing algorithms* for partitioning (i.e., assigning runnables to tasks) and discussed a couple of improvements to ease the application of the partitioning tool in practice (e.g., visualization frameworks).

Next, Chapter 3 described the current state of the mapping tool. It was enhanced in Amalthea4public according to new requirements and experiences made in industrial practice: a new approach to mapping based on genetic algorithms was investigated to address more than one quality attribute and the hardware model (a basis for mapping) was extended in several directions based on the feedback of the Amalthea4public partners.

Chapter 4 introduces the idea to widen the scope of partitioning and mapping from a single ECU to ECU networks. The MechatronicUML offers a domain-specific language called ASL (based on the OMG Object Constraint Language) for specifying allocation constraints. The ASL is extended by a set of conditions and allocations constraints defined on theses conditions to address specific aspects of ECU networks (e.g.,communication latency).

Chapter 5 illustrated how safety concepts as defined in ISO26262 (e.g., safety goals, requirements) affect the WP2-relevant concepts of the AMALTHEA software model. An example case was presented, to enhance partitioning and mapping by adding constraints derived from safety concepts (e.g., property constraints on communication interfaces such as CAN vs. FlexRay).

In Chapter 6 we discussed the current protocols for accessing shared resources by synchronized tasks. A new resource management concept for Amalthea4public called Test Delta based Runnable Rescheduling (TDRR) was presented, which makes use of the AMALTHEA software model (e.g., execution orders of runnables, label accesses) to calculate a Runnable schedule that reduces busy waiting times.

Finally, Chapter 7 presents the latest developments on the WP2 case studies. One area of activity was the implementation of selected concepts of the MCAPI to enable communication between cores for the NXP MPC5668G (a dual core CPU for embedded automotive applications). A second area of activity was the evaluation Linux as a possible platform for case studies, as Linux-based controllers are wide-spread and cheap.

The final deliverable D2.3 will focus on the prototypical implementation and validation of selected techniques presented in D2.1 and D2.2.

Bibliography

[1] Amalthea4public Project Partners: *Deliverable: D3.1, Analysis of state of the art V&V techniques.* May 2015

[2] Amalthea4public Project Partners: *Deliverable: D2.1, Concept for Requirements and Architectural Models for Multicore Systems.* Feb 2016

[3] Aleti, A. ; Buhnova, B. ; Grunske, L. ; Koziolek, A. ; Meedeniya, I.: Software Architecture Optimization Methods: A Systematic Literature Review. In: *Software Engineering, IEEE Transactions on* 39 (2013), May, Nr. 5, S. 658–683. – ISSN 0098-5589

[4] Alfranseder, Martin ; Deubzer, Michael ; Justus, Benjamin ; Siemers, Christian: An Efficient Spin-Lock Based Multi-core Resource Sharing Protocol. In: *IEEE International Performance Computing and Communications Conference* (2014)

[5] AMALTHEA: Deliverable D3.3 - Specification of a Trace Format and Target Platform Case Study. April 2013. – Forschungsbericht. Deliverable Report

[6] AMALTHEA: *Deliverable D3.4: Prototypical Implementation of Selected Concepts.* April 2014

[7] AMALTHEA4public: Deliverable D2.1 - Concept for Requirements and Architectural Models for Multicore Systems. February 2016. – Forschungsbericht. Deliverable Report

[8] AMALTHEA4public: Deliverable D2.1 - Concept for Requirements and Architectural Models for Multicore Systems. February 2016. – Forschungsbericht. Deliverable Report

[9] Annamalai ; Obeo ; Pierre-Charles, David u. a.: *Sirius starter tutorial.* 2016. – URL `https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial`

[10] Arabnejad, Hamid ; Barbosa, Jorge G.: List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. In: *IEEE Trans. Parallel Distrib. Syst.* 25 (2014), März, Nr. 3, S. 682–694. – ISSN 1045-9219

[11] Baker, T P.: A Stack-Based Resource Allocation Policy for Realtime Processes. (1990)

[12] Block, Aaron ; Leontyev, Hennadiy ; Brandenburg, B ; Anderson, James H.: A Flexible Real-Time Locking Protocol for Multiprocessors. (2007), Nr. Rtcsa. ISBN 0769529755

[13] Bodis, Attila: Bin Packing with Directed Stackability Conflicts. In: *Acta Universitatis Sapientiae, Informatica* 7 (2015), Nr. 1, S. 31–57

[14] Brandenburg, B ; Anderson, James H.: The OMLP Family of Optimal Multiprocessor Real-Time Locking Protocols. In: *Design Automation for Embedded Systems* (2012), S. 1–55

[15] Brandenburg, Björn B. ; Anderson, James H.: Optimality Results for Multiprocessor Real-Time Locking. In: *Proceedings of the 31st IEEE Real-Time Systems Symposium RTSS, San Diego, California, USA*, 2010, S. 49–60

[16] Chen, Min-ih ; Lin, Kwei-jay: Dynamic Priority Ceilings : A Concurrency Control Protocol for Real-Time Systems. In: *Real-Time Systems* Bd. 2, Kluwer Academic Publishers, 1990, S. 325–346. – ISSN 1573-1383

[17] Cheng, Albert M K. ; Ras, James: The Implementation of the Priority Ceiling Protocol in Ada-2005. (2005)

[18] Chinneck, John W.: Practical optimization: a gentle introduction. In: *Systems and Computer Engineering), Carleton University, Ottawa. http://www. sce. carleton. ca/faculty/chinneck/po. html* (2006). – URL `http://www.sce.carleton.ca/faculty/chinneck/po/TitlePageAndTOC.pdf`

[19] Coley, David A.: *An Introduction to Genetic Algorithms for Scientists and Engineers*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1998. – ISBN 9810236026

[20] Consotrium, AUTOSAR: *Overview of Functional Safety Measures in AUTOSAR*. 2015. – URL `https://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf`

[21] Cordes, Daniel A.: *Automatic parallelization for embedded multi-core systems using high level cost models*, TU Dortmund University, Dissertation, 2013

[22] Dziwok, Stefan ; Gerking, Christopher ; Becker, Steffen ; Thiele, Sebastian ; Heinzemann, Christian ; Pohlmann, Uwe: A Tool Suite for the Model-Driven Software Engineering of Cyber-Physical Systems. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, November 2014 (FSE 2014), S. 715–718. – ISBN 978-1-4503-3056-5

[23] Emde, Carsten ; Gleixner, Thomas: Quality assessment of real-time Linux. In: *Boards & Solutions / ECE Magazine* (2011). – URL `https://www.osadl.org/uploads/media/ECE-2011-09.pdf`

[24] Faragardi, Hamid R. ; Sandstr, Kristian ; Nolte, Thomas: An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-core Systems. In: *7'th International Symposium on Telecommunications* (2014), S. 41–48. ISBN 9781479953592

[25] Foster, Ian: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0201575949

[26] Frey, Patrick: *A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems Dissertation*, Ulm University, Dissertation, 2010

[27] Gai, Paolo ; Natale, Marco D. ; Lipari, Giuseppe ; Superiore, Scuola ; Anna, Sant ; Ferrari, Alberto ; Gabellini, Claudio ; Marceca, Paolo: A comparison of MPCP and

MSRP when sharing resources in the Janus multiple-processor on a chip platform. (2003), S. 1–10

[28] Geismann, Johannes: *Multi-Core Execution of Safety-Critical Component-Based Software*. Zukunftsmeile 1, 33102 Paderborn, Germany, Software Engineering, Heinz Nixdorf Institute, Paderborn University, Master's Thesis, Dezember 2015

[29] Gomes, Renata M. ; Mauroner, Fabian ; Baunach, Marcel: Collaborative Resource Management for Multi-Core AUTOSAR OS. In: Halang, W.A. (Hrsg.) ; Spinczyk, O. (Hrsg.): *Betriebssysteme und Echtzeit, Informatik aktuell*. Springer-Verlag Berlin Heidelberg, 2015, S. 99–108. – ISBN 9783662486115

[30] Graphviz consortium: *Graphviz - Graph Visualization Software*. 2016. – URL `http://www.graphviz.org`

[31] Holtmann, Jörg ; Bernijazov, Ruslan ; Meyer, Matthias ; Schmelter, David ; Tschirner, Christian: Integrated and iterative systems engineering and software requirements engineering for technical systems. In: *Journal of Software Evolution and Process* (2016), Mai

[32] Ilavarasan, E. ; Thambidurai, P.: Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments. In: *Journal of Computer Sciences* Bd. 3, 2007, S. 94–103

[33] ISO: *ISO 26262 - Road vehicles — Functional safety*. Juli 2009

[34] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 3 Concept phase*. Juli 2009

[35] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 4 Product development at the system level*. Juli 2009

[36] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 6 Product development at the software level*. Juli 2009

[37] Krawczyk, Lukas ; Wolff, Carsten ; Fruhner, Daniel: *Automated Distribution of Software to Multi-core Hardware in Model Based Embedded Systems Development*. S. 320–329. In: Dregvaite, Giedre (Hrsg.) ; Damasevicius, Robertas (Hrsg.): *Information and Software Technologies: 21st International Conference, ICIST 2015, Druskininkai, Lithuania, October 15-16, 2015, Proceedings*, Springer International Publishing, 2015. – URL `http://dx.doi.org/10.1007/978-3-319-24770-0_28`. – ISBN 978-3-319-24770-0

[38] linutronix GmbH: *LINUX und Echtzeit - Eine Übersicht prinzipieller Lösungsansätze*. 2011. – URL `https://www.linutronix.de/uploads/images/PDF/WP_2011_Linux_+_Echtzeit_prinzipielle_Ansaetze_V1_0.pdf`

[39] Naveh, Barak ; Contributors: *JgraphT website*. June 2016. – URL `http://jgrapht.org`

[40] Nocedal, Jorge ; Wright, Stephen: *Numerical Optimization*. Springer, 2006. – ISBN 0387303030

[41] : *oj! Algorithms Webpage*. `http://ojalgo.org/`. June 2014. – URL `http://ojalgo.org/`

[42] PlantUML consotrium: *Drawing UML with PlantUML - Language Reference Guide*. 2016. – URL http://plantuml.com/PlantUML_Language_Reference_Guide.pdf

[43] Rajkumar, Ragunathan: *Synchronization in Multiple Processor Systems*. S. 61–118. In: *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Boston, MA : Springer US, 1991. – ISBN 978-1-4615-4000-7

[44] Ripoll, Ismael ; Ballester-ripoll, Rafael: Period Selection for Minimal Hyperperiod in Periodic Task Systems. In: *IEEE Transactions on Computers* Bd. 62, IEEE Computer Society, September 2013, S. 1813–1822

[45] Sha, Lui ; Rajkumar, Ragunathan ; Lehoczky, John P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In: *IEEE Transactions on Computers* 39 (1990), Nr. 9, S. 1175–1185

[46] Suhl, Leena ; Mellouli, Taïeb: *Optimierungssysteme*. 3., korr. u. aktualisierte Aufl. Berlin [u.a.] : Springer Gabler, 2013 (Springer-Lehrbuch). – ISBN 9783642389368

[47] The Multicore Association: *Multicore Communications API (MCAPI) Specification V2.015*. http://www.multicore-association.org/workgroup/mcapi.php. 2011

[48] Topcuoglu, Haluk ; Hariri, Salim: Performance-Effective and Low-Complexity. In: *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS* 13 (2002), Nr. 3, S. 260–274

[49] Ward, Bryan C. ; Anderson, James H.: Supporting Nested Locking in Multiprocessor Real-Time Systems. In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, IEEE Computer Society, 2012, S. 223–232

[50] Wilhelmstötter, Franz: *Jenetics is an advanced Genetic Algorithm, respectively an Evolutionary Algorithm, library written in modern day Java*. July 2016. – URL http://jenetics.io

[51] Zhang, Yumin ; Hu, Xiaobo S. ; Chen, Danny Z.: Task Scheduling and Voltage Selection for Energy Minimization. In: *Proceedings of the 39th annual Design Automation Conference*, ACM, 2002, S. 183–188. – URL http://dl.acm.org/citation.cfm?id=513966