AMALTHEA

(ITEA 2 – 09013)

Model Based Open Source Development Environment
for Automotive Multi-Core Systems

Deliverable: D 3.4

Prototypical Implementation of Selected Concepts

Work Package: 3
Target Mapping

Task: 3.3
Development of Scheduling Analysis and Partitioning/Mapping Support Tools

Document type:	Deliverable	**Classification:**	**Public**
Document version:	Final	**Contract Start Date:**	01.07.2011
Document Preparation Date:	30.04.2014	**Duration:**	30.04.2014

ITEA 2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

EUREKA

History

Rev.	Content	Resp. Partner	Date
1.0	Final Version	FH Dortmund	2014-04-30
0.9	Draft Version	C-LAB	WIP

Final Approval	Name	Partner
Review Task Level	WIP	WIP
Review WP Level	Kamsties	FH Dortmund
Review Board Level	Topp	Bosch

Contents

List of Figures

List of Tables

Executive Summary

AMALTHEA is an ITEA 2 funded project that is developing an open and expandable tool platform for automotive embedded-system engineering based on model-driven methodology. Key features include the support for multi-core systems combined with AUTOSAR compatibility and product-line engineering. The resulting AMALTHEA Tool Platform is distributed under the Eclipse Public License.

This document is the fourth and final deliverable of Work Package 3 "Target Mapping" of AMALTHEA. It is produced within Task 3.3 "Development of Scheduling Analysis and Partitioning/Mapping Support Tools" and Task 3.4 "Integration and Development of Code Generators for Embedded Targets".

Content. This deliverable D 3.4 "Prototypical Implementation of Selected Concepts" provides an overview of the AMALTHEA toolchain platform and introduces the HVAC demonstrator built in WP3. The main part of this deliverable discusses the tool support for the main technical activities, namely modeling, partitioning, mapping, building, and tracing. Each tool is described by the concepts behind the tool, a discussion of the prototypical implementation, and examples of concept/tool application drawn from the HVAC demonstrator. Special focus is on partitioning, mapping, and tracing.

Intended readership. The target audience of this document is the project sponsors and project members of the AMALTHEA project. Moreover, this document addresses also *project-external readers who are interested in software development for multi-core CPUs in the embedded system domain, in particular in the automotive domain.* For this purpose, this document also includes some introductory material about AMALTHEA and can be read without referring to other documents.

Overview. The document is structured in the following main chapters:

Chapter 1 "Introduction" provides a short introduction to the results of WP3 and the relation to other parts of the AMALTHEA project.

Chapter 2 "Overview on Toolchain" provides an overview on the AMALTHEA toolchain platform.

Chapter 3 "Demonstrator" introduces the HVAC (Heating, Ventilation, and Air Conditioning) demonstrator, in particular, the use case, the technical architecture, and its prototypical HMI (human machine interface).

Chapter 4 "Requirements Engineering and Variability Modeling" provides a brief overview on the respective activities, which are prerequisite to target mapping, but subject of other work packages (WP1 and WP2, respectively).

Chapter 5 "Architectural and Behavioral Modeling" describes the modeling of the architecture and behavior of an embedded multi-core system. The HVAC system is used as an example for the various steps from interface and components specification to behavior modeling and model transformation.

Chapter 6 "Partitioning" introduces our abstract graph-based approach of modeling multi-core software and describes the different partitioning algorithms and their implementation.

Chapter 7 "Mapping" illustrates the concept behind the mapping plugin and contains details about the implementation and its algorithms.

Chapter 8 "Building" describes how the outputs of the previous steps are processed in order to build an executable, which runs on a multi-core target platform. Two main steps are addressed in AMALTHEA. First, generating code from the architectural models, and second, generating an OSEK configuration.

Chapter 9 "Tracing" provides a specification of the AMALTHEA Hardware Trace Format (HTF) and describes, how a trace is generated on the target platform and converted into the OT1 format selected by WP3 (see Deliverable 3.3), and finally visualized using the Timing Architects Toolsuite.

Chapter 10 "Conclusion" closes this deliverable.

1 Introduction

AMALTHEA is an ITEA 2 funded project that is developing an open and expandable toolchain for automotive embedded-system engineering based on model-driven methodology. Key features include the support for multi-core systems combined with AUTOSAR compatibility and product-line engineering. The resulting AMALTHEA Tool Platform is distributed under the Eclipse Public License.

The AMALTHEA Tool Platform consists of a *basic infrastructure*, that is a comprehensive data model and a generic execution environment (see [3]) and a set of open-source tools forming a seamless *toolchain* (described in this document). The tool platform covers all activities for developing embedded software for multi-core systems (see Figure 1.1).

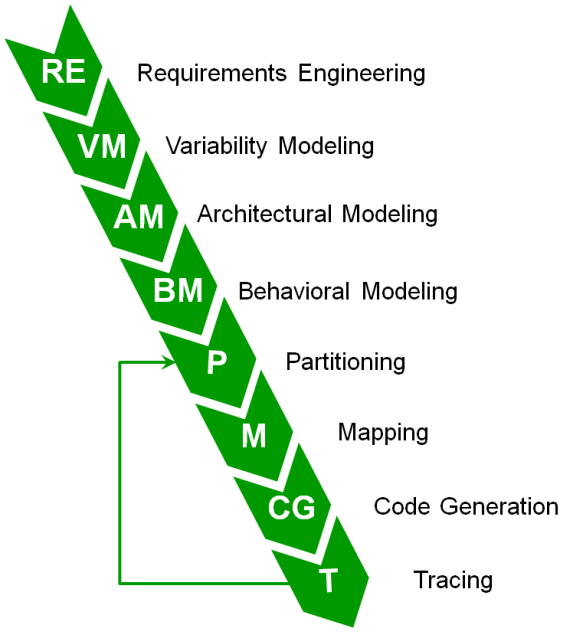

Figure 1.1: Overview on AMALTHEA Tool Platform

During the requirements engineering (RE) phase, the requirements of the multi-core system are gathered in an informal way. In variability modeling (VM), the software and hardware variability is defined. Architectural modeling (AM) focuses on the identification and representation of components and their interconnections. The behavior of each component is specified in behavioral modeling (BM), different modeling techniques can be applied depending on the nature of the component (e.g., Matlab Simulink, Statecharts,

or plain C code).

Partitioning (P) and mapping (M) are specific activities required for multi-core systems. In partitioning, the system (modeled in the previous activities) is divided into tasks. This step is automated and results in an optimal solution regarding particular quality characteristics, which can be influenced by a developer. Mapping concerns the distribution of tasks to cores, which is non-trivial as heterogeneous multi-core processors are usually used in the embedded domain. Mapping is automatized as well and seeks to optimize for instance energy-efficiency. Code generation (CG) is straight forward from a theoretical viewpoint. From a practical viewpoint, collecting source code from different sources (architectural modeling, behavioral modeling, OS configuration) and feeding it to a compiler is a bit of a challenge.

Finally, tracing (T) is the activity of collecting data about the execution of the software on the multi-core hardware. The data acquired from the hardware allows to improve the partitioning and mapping of the software, as these activities rely on informed guesses first. Trace data provides a more solid base of information.

The support of the activities of the tool platform was distributed over the different work packages. Work Package 1 was concerned with requirements engineering, Work Package 2 provided a tool for variability modeling. The focus of Work Package 3 (WP3) was on the remaining six activities: AM, BM, P, M, CG, and T. Work Package 4 delivered the basic infrastructure of the whole tool platform.

The following chapter provides an introduction to the AMALTHEA Tool Platform.

2 Overview on Tool Platform

This chapter provides an overview on the AMALTHEA Tool Platform. Furthermore, it describes an instantiation of the AMALTHEA Tool Platform, which is extended by further tools and used as an example toolchain throughout this deliverable.

2.1 Overview

The AMALTHEA Tool Platform, depicted in Figure 2.1, supports all development activities described in Chapter 1. It is built on Eclipse. From a logical point of view, the AMALTHEA Tool Platform consists of *tools* and *models*. From a technical point of view, tools and models are so-called *features* in Eclipse. An Eclipse feature describes a list of plug-ins and other features, which can be understood as a logical unit.[1]

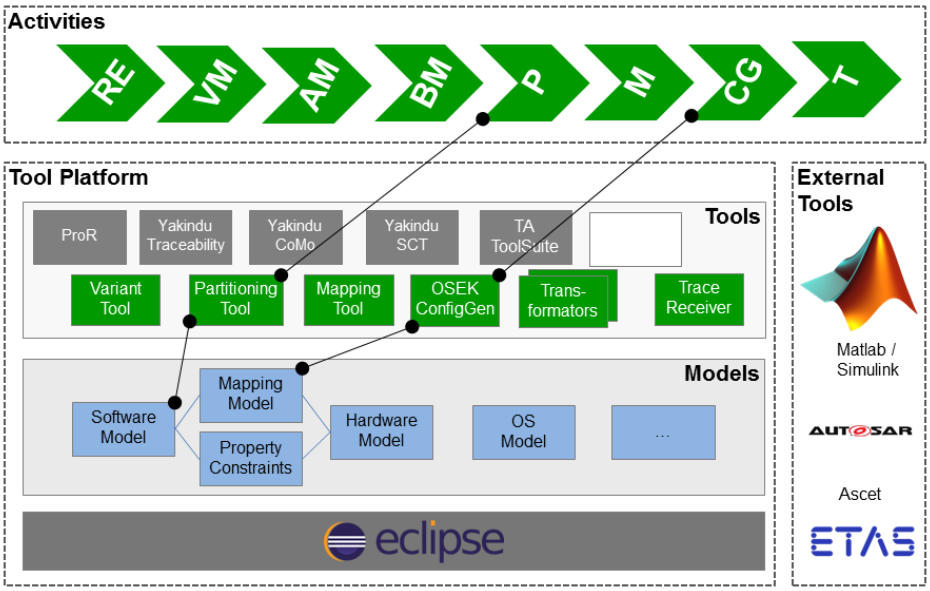

Figure 2.1: Overview on AMALTHEA Tool Platform

The **tools** of the platform support particular activities. For example, the OSEK Configuration Generator is used in code generation. We separate between tools developed in AMALTHEA (green boxes) and additional tools (grey boxes), which are provided by

[1] Please refer to the deliverables of Work package 4 for a detailed discussion of the technical view, the focus of this chapter is on the logical view of the AMALTHEA Tool Platform.

partners (independent of the AMALTHEA project) or by third-parties. All AMALTHEA tools are published as open source under the Eclipse Public License, additional tools have various licenses (see Section 2.1.2).

The AMALTHEA Tool Platform provides a set of **models** describing different aspects of a system. These models are *descriptive*; they are intended as *abstractions* of usually proprietary, prescriptive models. For example, the hardware model describes properties of the hardware independent of a specific implementation language such as SystemC or VHDL. The hardware model is *intended to inform* several activities, e.g., mapping of tasks to cores, but it is *not intended to synthesize* hardware. The models are available in EMF from the AMALTHEA project page [7] and more information is available in the deliverables of Work Package 4. Common to all AMALTHEA tools is that they work on the AMALTHEA models as opposed to the additional tools, which do not require AMALTHEA models.

An external tool is a tool that does not have Eclipse as a basis. That is, it cannot be integrated into the AMALTHEA Tool Platform to form a homogeneous toolchain. An exporter or importer is required to use such tools with the AMALTHEA Tool Platform. At the time of writing, popular external tools such as Matlab/Simulink or ETAS/Ascet are supported by importer/exporter or an OSLC integration, respectively. The following subsections describe the models and tools briefly.

2.1.1 Models

The AMALTHEA Tool Platform provides a set of models. This section briefly describes the purpose of selected models.

The **software model** is used to evaluate different approaches for the distribution of software and as a basis for the use of commercial (scheduling) simulation tools and covers the abstraction of the software. Moreover, the software model informs the build process in order to produce source code and executables. For this purpose it is necessary to have more information about the software, like data structures, mapping information, and target platform data. The basic and most important abstraction is called *runnable* and represents a specific execution unit, which consumes calculation power and accesses labels. For instance, a runnable can represent a single operational unit (such as MAC, Add, Shift operations) from a Matlab control flow model.

The **hardware model** is used to describe hardware systems that consist of ECUs, microcontrollers, cores, memories, networks, additional peripherals and more.

The **constraints model** it is generated by the partitioning plugin and contains different kinds of constraints. The basis is formed by so called *runnable sequence constraints* (RSCs) that can be used to define a required order for the runnables. Moreover, the affinity constraints define constraints for the purpose of mapping runnables, processes, and schedulers. Furthermore, timing constraints provide the restriction of the time span between events or the duration of event chains [2].

The **operating system model** mainly provides information on how access is given to certain system resources. Therefore, the concepts of scheduling, buffering, and semaphores are supported. A *scheduler* controls the execution of processes on a CPU. There can be

multiple schedulers in an operating system model. Each scheduler can manage one or more hardware cores. The cores are mapped to the scheduler via the mapping model. Each scheduler has one scheduling algorithm and a scheduling unit. The scheduling unit can be either a hardware scheduling unit or a software scheduling unit.

The **mapping model** provides information about the mappings and allocations between elements of hardware and software models. In particular, this model contains associations between schedulers and executable software, schedulers and cores, and data and memories. Allocations assign executable software, cores, runnables or processes to scheduler and mappings assign labels, sections or software to memories.

The **property constraints model** limits the design space by providing information about the specific hardware properties or features required by software elements. Such information is mandatory within the mapping process, since software elements can only be executed on a hardware unit that provides the specific properties and features required by the software element. Within the AMALTHEA HVAC demonstrator for instance, property constraints arise from the fact that some software entities require FPU features that only one core on the hardware platform (Freescale MPC 5668G) provides. Similar to the mapping model, such constraints can address allocations to cores or mapping to memories.

2.1.2 Tools

The following subsections introduce tools that can be used within the AMALTHEA Tool Platform. We separate between tools developed in AMALTHEA and additional tools (supplied by partners or third-parties), and external tools (that do built on Eclipse).

AMALTHEA Tools

The Variant Modeler was developed by Work Package 2. It supports the development of hardware and software variability models [6]. Figure 4.2 shows the variant modeler within the AMALTHEA Tool Platform and illustrates the HVAC variability model, whereas some mandatory and some optional features are modeled.

The following tools were developed in WP3:

- Partitioning tool (see Chapter 6)
- Mapping tool (see Chapter 7)
- OSEK Configuration Generator tool (see Chapter 8.2.2)
- System Generator tool (not shown in Figure 2.1, see Chapter 8.2.1)
- Trace Receiver tool (see Chapter 9.2.2)
- Transformators
  - CoMo Transformator: Import of simplified component model into the AMALTHEA software model. More detail is provided in Chapter 5.

– Trace Converter: Converts a HTF trace into an OT1 trace. More detail is in Chapter 9.2.3.

Additional Tools

There are a couple of external, third-party tools available that can be added to the AMALTHEA Tool Platform. In the following, we name those that were used in the AMALTHEA project. Others tools can be added as required.

ProR is a tool for requirements engineering, supporting the ReqIF Standard natively. It is built for extensibility and provides traceability between requirements and formal models. Requirements engineering is mandatory in most development processes and describes the functionality of the product often provided by the OEM (or product manager). Further, it defines what functions the product must be capable of, how different components work together, how safety is addressed, how constraints are met and many more. Figure 4.1 shows the ProR tool with some HVAC requirements.

YAKINDU is a toolsuite based on Eclipse developed by itemis AG, which consists of open-source and commercial tools. Several tools were used within AMALTHEA.

YAKINDU Traceability[2] is used to establish traceability links between the various development artifacts. It shows that AMALTHEA models also establish a common ground for tracing all involved artifacts.

YAKINDU Components Model (CoMo) is a tool to model the architecture of a system by components, connections between components, and by the declaration of a component's behavior.

YAKINDU Statechart Tools (SCT)[3] are open source and provide various features for statechart based modeling, simulation, verification, and code generation. Figure 2.2 shows a part of the HVAC statechart model within the YAKINDU SCT editor.

Timing Architects Toolsuite[4] supports partitioning and mapping (simulation).

External Tools

External tools which are supported by AMALTHEA:

- Matlab Simulink for Behavioral Modeling. A respective importer is available to extract the information required for partitioning and mapping from Simulink models (see D3.3).
- ERIKA Enterprise[5]: If the AMALTHEA Tool Platform is used for development, a compiler toolchain for a specific embedded target is needed. AMALTHEA supports the ERIKA Enterprise toolchain for OSEK based systems.

More extensions were undertaken by other work packages:

[2] http://www.yakindu.com/traceability/
[3] http://www.statecharts.org/
[4] http://www.timing-architects.com/ta-tool-suite.html
[5] http://erika.tuxfamily.org

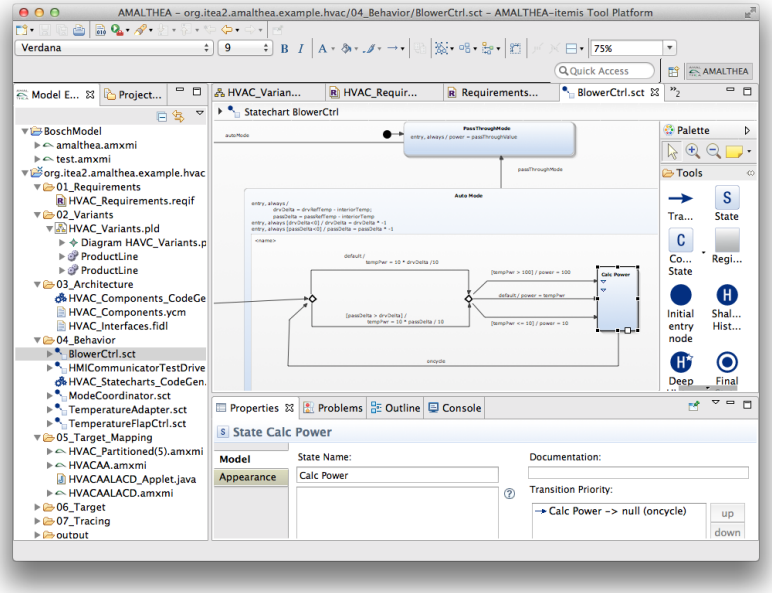

Figure 2.2: YAKINDU SCT within the AMALTHEA Tool Platform

- ETAS: ASCET (WP4)
- Timing Architects: TA Toolsuite (WP4)
- University of Oulu: Requirements Tool (WP1)
- Bosch: AMALTHEA Tool Platform for motor control unit (WP4)
- Tofas: EV Cooling System Control (WP5)

2.2 Example Instantiation of Tool Platform

The AMALTHEA Tool Platform can be instantiated with different tools for different purposes. We separate two main purposes that differ in how far the AMALTHEA toolchain replaces the current OEM/supplier toolchain:

- *Analysis* - a commercial toolchain is in place, e.g., based on MATLAB and AUTOSAR. An export into AMALTHEA models is done for the purpose of using the analysis capabilities of the AMALTHEA toolchain (partitioning, mapping).
- *Development* - AMALTHEA tools form the complete development toolchain.

The AMALTHEA Tool Platform can be instantiated with AMALTHEA tools and external, open source tools to provide a *complete open source solution* to the development of embedded multi-core systems.

Commercial tools may replace the open source tools if more functionality is required. Figure 2.3 shows that external tools and AMALTHEA tools can be combined into a seamless toolchain.

The specific selection of tools shown in Figure 2.3 was done with respect to an open case study of an HVAC system (see Chapter 3). Thus, this instantiation employs mainly open source tools.

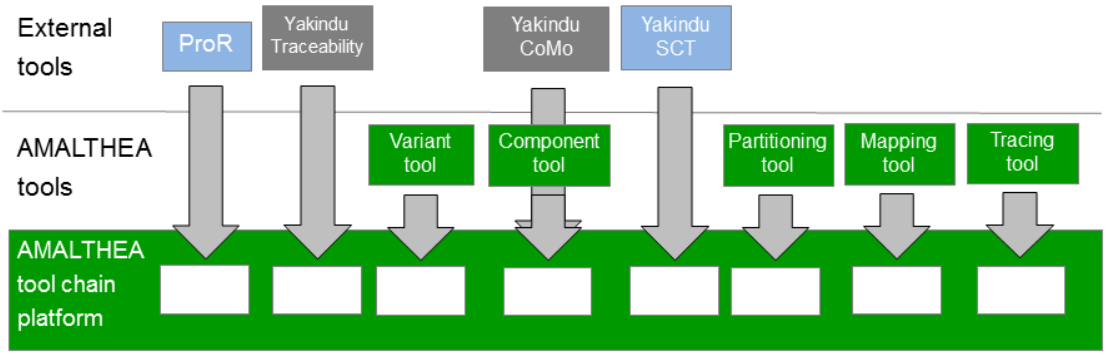

Figure 2.3: AMALTHEA toolchain instantiated for HVAC case study

ProR was chosen as a pragmatic requirements management tool, which integrates well into Eclipse. Requirements are outside the AMALTHEA data model, thus an importer is not required. Figure 4.1 shows a screenshot of the ProR tool.

The ProR requirements are the starting point of traceability, which is maintained with YAKINDU Traceability.

The next activity is to model the variability in case the system under consideration is actually a product line. Figure 4.2 on page 14 shows a variability model. The special focus within AMALTHEA is on the separation of software and hardware variability. More details are described in Deliverable 2.5.

YAKINDU CoMo and SCT (Statecharts) are responsible for architectural and behavioral modeling, respectively. All other tools are AMALTHEA tools were developed in WP3.

3 Demonstrator

This chapter introduces the HVAC use case, which was provided by the AMALTHEA partner BHTC, describes the technical architecture, and outlines the human-machine interface (HMI) of the HVAC.

3.1 HVAC Use Case

In the context of this deliverable, a demonstrator and its corresponding use case must show the feasibility of the AMALTHEA approach on the basis of an exemplary toolchain. For this purpose, the use case has to be complex enough to require the steps and appropriate tools of a typical software development process (requirements engineering, variability modeling, architectural modeling, etc.) without being too complex to obscure what really should be shown: the seamless functioning of the toolchain itself.

As a target platform BHTC provided a simplified version of an automotive HVAC (Heating, Ventilation and Air Conditioning) set up. Since the demonstrator should be portable and ready to be used in an educational environment, the *Air Conditioning* part (no need for a cooling circuit) and the *Heating* part (less demands on power supply) are omitted. The remaining *Ventilation* part with its control of air flow and air distribution still posts enough challenges. The remaining functional parts of this setting are

- a touch display, mimicking the control panel of a climate control ECU,
- a temperature sensor, representing the measurement of the vehicle's internal temperature,
- a blower, providing the air volume flow through the HVAC system,
- several flaps, controlling the distribution of the air flow inside the HVAC system,
- the evaluation board comprising a dual core processor for emulating the ECU, targeted by the toolchain.

The physical installation and setting will be described in more detail further below in this chapter.

Overall, this set up provides possibilities for

- *Requirements Engineering* for different levels of detail (functionality),
- *Variability Modeling* regarding
  - user interface (touch display internal to the ECU or data via CAN bus),

  - control of flaps (directly via LIN bus or data via CAN bus),

- *Architectural Modeling*, comprising components for
  - HMI communication, bus communication, reading sensor data, and actuator control, and
  - the higher-level HVAC application,
- *Behavioral Modeling*, implementing the functionality
- *Partitioning*, the definition of tasks and optimal assignment of runnables to tasks,
- *Mapping*, the optimal distribution of tasks onto different cores,
- *Building*, generating and compiling code, and
- *Tracing*, which is the recording of timing events of the software execution on the target.

The development steps from modeling (architectural and behavioral) down to tracing will be elaborated in the following chapters of this deliverable.

It has to be kept in mind that the real demonstration object is not the HVAC use case, but the exemplary toolchain by which means it is implemented.

3.2 Architecture

The demonstrator consists of several devices, Figure 3.1 shows the technical architecture. At the heart is a Freescale MPC5668G based evaluation board. The MPC5668G is a dual-core processor for automotive applications offering several communication interfaces such as SPI, LIN, and CAN. The HMI is implemented on an Android tablet, which is linked to a tablet connection device. The tablet connection is a device, which translates between WLAN and SPI. The blower of the HVAC device is controlled over PWM. The flaps are controlled over LIN. The evaluation board is connected to a temperature sensor over a typical analog-digital converter.

The air flow within the HVAC device is shown 3.2. Fresh air is taken in by the blower. The fresh air is dried and cooled down by an evaporator. A flap (“blend door”) circumvents the heater (which not used in our case study). The air flow is finally distributed to several channels ("defrost", "vent", "floor") depending on the position of several other flaps.

3.3 HMI

The HMI is an application developed for Android tablets (screenshot shown in Figure 3.3). It allows a person to control the HVAC system and it displays the actual mode and other information like indoor temperature.

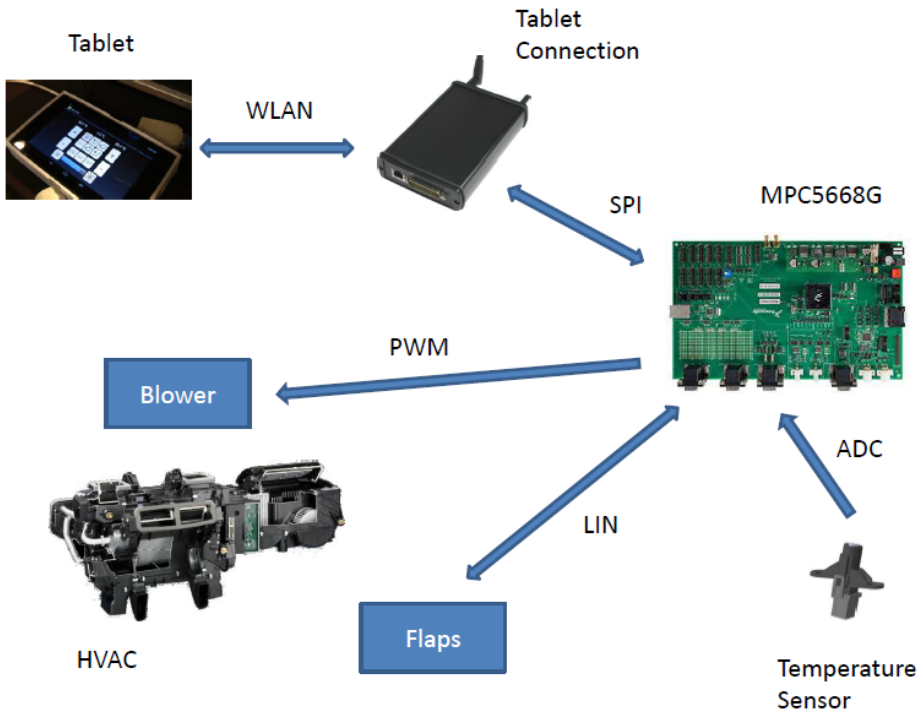

Figure 3.1: Architecture of HVAC Demonstrator

The application starts in *automatic* mode. The user can select between *defrost* mode, *vent* mode, a *mixed* mode from defrost and floor, automatic mode, bi-level mode (which is a combination of vent and floor mode), and floor-only mode. In addition, the recirculation and air condition can be activated. It is also possible to switch off the full system. In defrost mode, the blower runs on full power. The modes vent and floor set the respective door to the targeted level, depending on the mode. In automatic mode, the system defines the right level depending on the environment.

The temperature for driver and assistant driver can be chosen separately. The speed of the blower can be chosen and the actual level from the control unit is displayed in a bar diagram.

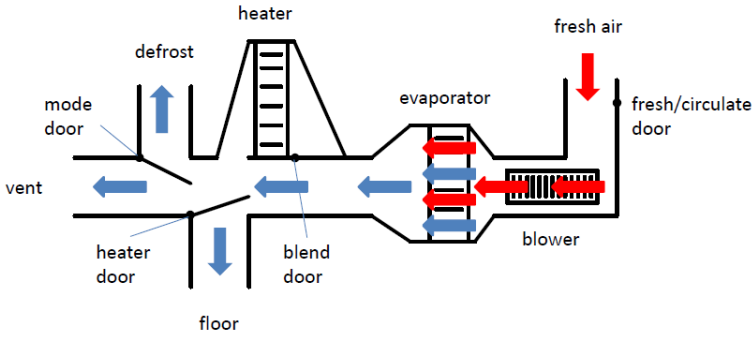

Figure 3.2: Air Flow in HVAC Device

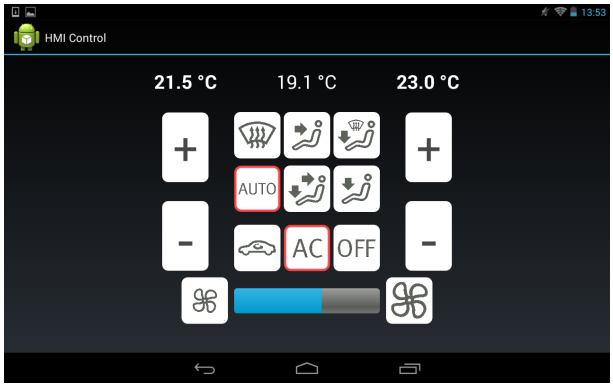

Figure 3.3: HMI Control Application

4 Requirements Engineering and Variability Modeling

This chapter describes the first development activities supported by the AMALTHEA tool platform. The discussion is brief, because Requirements Engineering (RE) and Variability Modeling (VM) are in the focus of Work Package 1 and 2, respectively.

The HVAC case study was carried out beginning with the very first activity supported by the AMALTHEA Tool Platform, namely requirements engineering. Figure 4.1 shows a screenshot of the HVAC requirements.

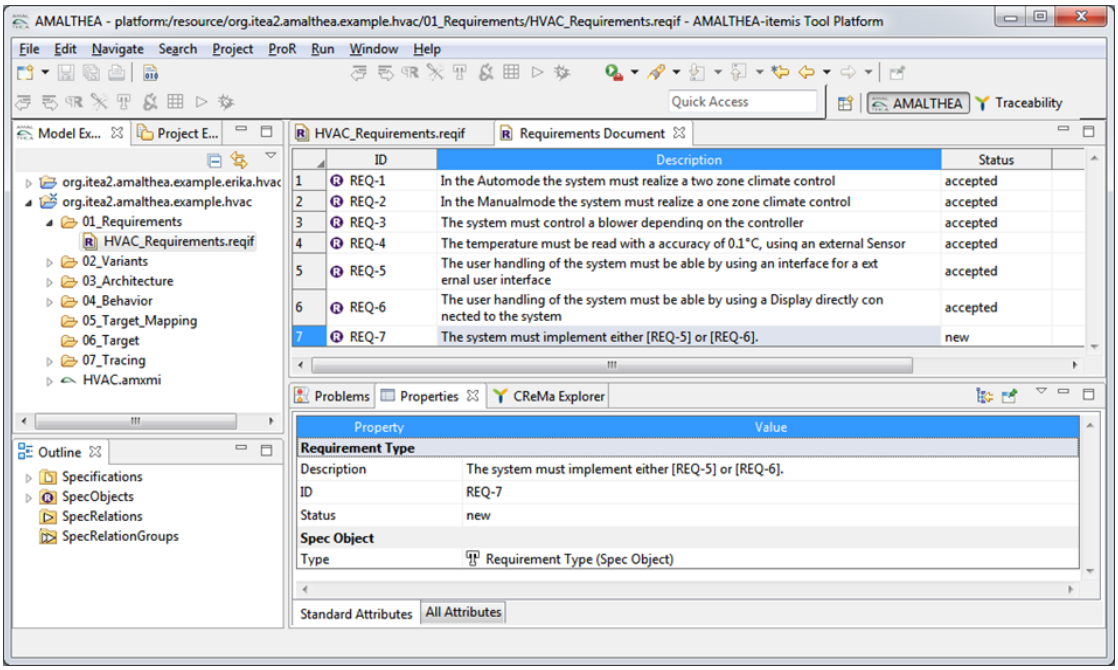

Figure 4.1: HVAC Requirements in ProR Tool

The next activity is to model the variability, because the system under consideration is actually a product line. Figure 4.2 shows a screenshot of the HVAC feature model. More details are described in Deliverable 2.5.

The next two steps concern Architectural Modeling and Behavioral Modeling and are described in the following chapter.

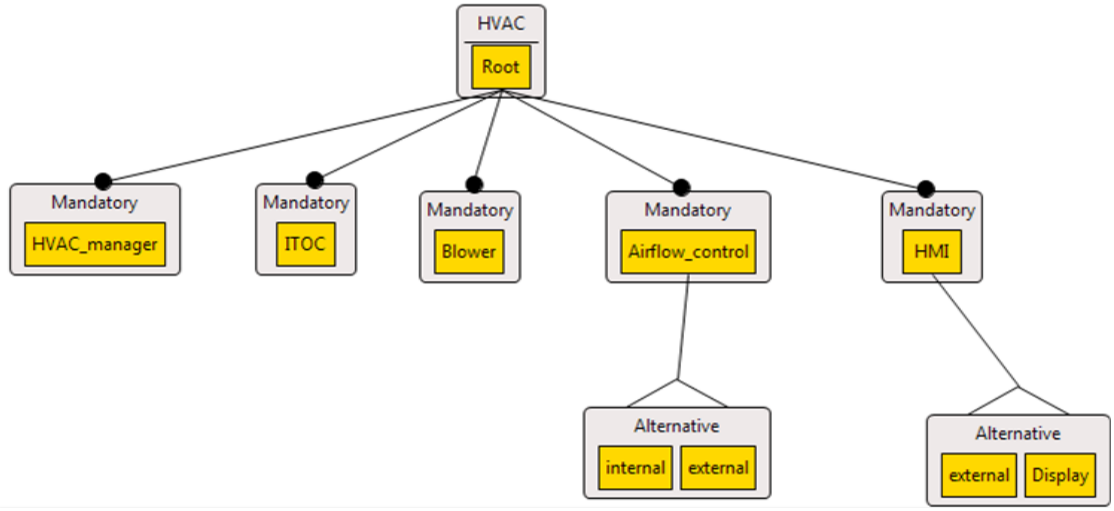

Figure 4.2: HVAC Feature Model in AMALTHEA Variability Tool

5 Architectural and Behavioral Modeling

In this chapter the modeling of the architecture and behavior of an embedded multi-core system is described. The HVAC system is used as an example for the various steps from interface and components specification to behavior modeling and model transformation. In the following, these steps are described in detail.

5.1 Specification of Interfaces

When decomposing a system into several components, interfaces serve as glue between the components. For the specification of interfaces, the Franca Interface Definition Language (IDL) is integrated into the AMALTHEA tool chain. In the HVAC system, interfaces are mainly used to specify data and events which are transferred between the various components. The data is stored in the attributes of interfaces whereas broadcasts are defined in interfaces to send events.

```
1  interface BlowerCtrl.power {
2    attribute UInt16 blwOutValue readonly noSubscriptions
3  }
4
5  interface TemperatureAdapter.Signals {
6    broadcast turnOn {}
7    broadcast turnOff {}
8  }
```

Listing 5.1: Interface Definitions in Franca IDL

In Listing 5.1, the interface BlowerCtrl.power between the blower of the HVAC system and its control is specified. The attribute blwOutValue of type UInt16 stores the power the blower will run with. It will be regularly read by the blower. This is why the attribute is set to be readonly. In the interface Temparature.Signals, two broadcasts are defined to turn the heating on and off. These two interfaces are just an excerpt of 10 interfaces specified for the HVAC demonstrator.

5.2 Specification of Components

For the HVAC demonstrator case study, a customized toolchain based on the AMALTHEA Tool Platform was used. The YAKINDU Component Model (CoMo) was added to the AMALTHEA Tool Platform to specify components and the overall system. It is currently developed as a commercial tool by the AMALTHEA project partner itemis. The HVAC demonstrator served as a use case for the development of YAKINDU CoMo. YAKINDU CoMo is one example for the dissemination of the AMALTHEA project.

```
component HVAC_Blower {
  port power requires BlowerCtrl.power

  behavior setPower
    ports power
    trigger every 100ms
}
```

Listing 5.2: Definition of Component HVAC_Blower

In Listing 5.2, the component for the HVAC blower is specified. It has a port that requires the interface BlowerCtrl.power. This port is called power. Furthermore, it declares the component's behavior setPower that reads the port power every 100ms and adjusts the blower's power to speed it up or slow it down.

The actual implementation of the HVAC_Blower component could be done with any tool, for example Matlab Simulink, or just manually in plain C. YAKINDU CoMo does not depend on a certain implementation method for the components. Nearly any kind of a component's implementation might be integrated into the component model. Currently, the only limitation is that the component's implementation must be linkable to a C-header that is generated from the component. In the embedded domain, this is usually no limitation. The C-header defines amongst other things an interface for the behavior. Anyhow, YAKINDU CoMo is designed to be modular, i.e. the code generation might be changed easily to integrate components in any other way.

```
component TemperatureAdapter {
  port ticks requires TemperatureAdapter.Ticks
  port mode  requires TemperatureAdapter.Signals
  port temp  provides Temperature

  behavior runCycle
    ports mode, temp, ticks
    trigger every 20ms
    pim
    with statechart -> TemperatureAdapter {
      ticks.commToggle -> temperatureSet
      ticks.minusTicks -> minusTicks
      ticks.plusTicks  -> plusTicks

      mode.turnOff -> off
      mode.turnOn  -> on

      temp.temp -> temp
    }
}
```

Listing 5.3: Definition of a Component with a Statechart

Listing 5.3 shows the component specification of the TemperatureAdapter. It has two ports that require two different interfaces and one port that provides a third one. The component specifies a behavior called runCycle that is triggered every 20ms. In this example, the behavior is not only declared but it is also linked to its implementation by a statechart. The statechart is also named TemperatureAdapter. Within the declaration of the behavior, attributes and broadcasts of the three ports are mapped to variables respectively events of the statechart. In the following, the statechart and its relation to the component is explained in detail.

5.3 Specification of the Component's Behavior

For the behavior specification of some HVAC components the YAKINDU Statechart Tools (SCT) were used. YAKINDU SCT is provided by itemis, too. Since it is open source, it is already integrated into the official AMALTHEA Tool Platform.

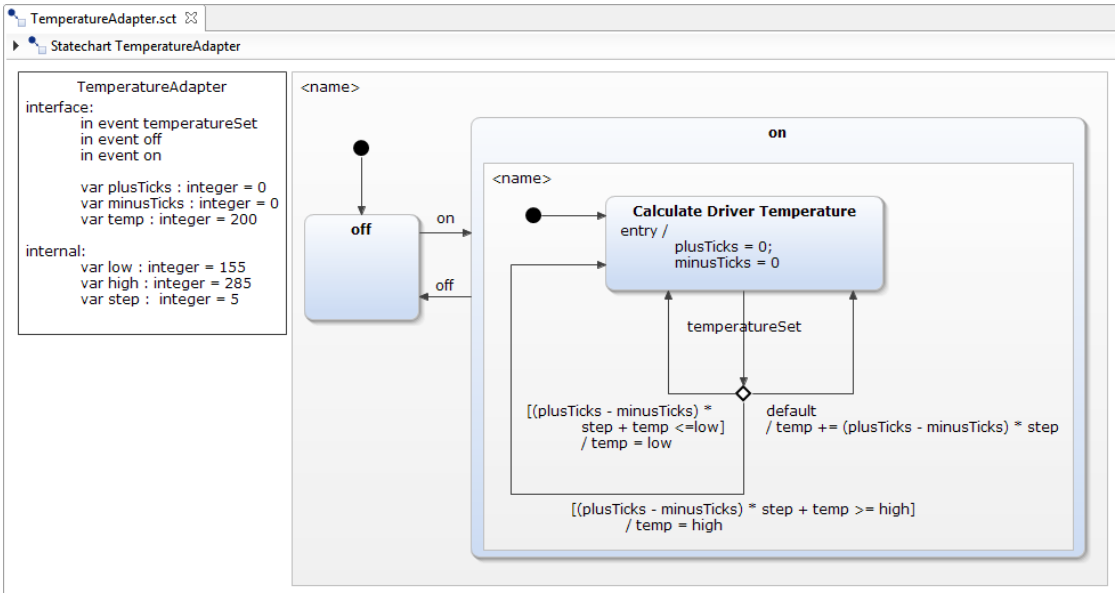

Figure 5.1: Statechart of the TemperatureAdapter Component

Figure 5.1 shows the statechart of the TemperatureAdapter. It is a pretty simple one with only two states and a region. Furthermore, as a special feature of the YAKINDU SCT, there is a declaration of the interface and of internal variables with types and default values. The interface of the statechart is the interesting part. It is actually linked to the component's specification (see Listing 5.3). For example, the events on and off are mapped to the broadcast messages turnOn and turnOff defined in the interface TemperatureAdapter.Signals which is required by the port mode.

The variable temp is calculated within the statechart. It is mapped to the port temp that provides the interface Temperature. This interface defines itself an attribute temp. Thus, the statechart receives data from other components via the variables plusTicks and minusTicks, calculates the variable temp, and provides its value to other components.

5.4 Specification of the System

Once the components are specified, the overall system needs to be composed from the components and their interfaces needs to be connected together. Listing 5.4 shows an excerpt of the HVAC system.

```
1  system HVAC_YSCT {
2    instance hmi:HMI_Communicator;
3    instance coordinator: ModeCoordinator {
4      blwMaxSteps := 10
5    };
6    instance drvTempAdapter: TemperatureAdapter;
7    instance drvTempFlapCtrl: TemperatureFlapCtrl;
8
9    instance passTempAdapter: TemperatureAdapter;
10   instance passTempFlapCtrl: TemperatureFlapCtrl;
11
12   instance blowerCtrl: BlowerCtrl;
13
14   instance hvacBlower: HVAC_Blower;
15
16   connect drvTempAdapter.temp with blowerCtrl.drvTempSet;
17   connect drvTempAdapter.temp with hmi.drvTempSet;
18   connect drvTempAdapter.temp with drvTempFlapCtrl.setTemp;
19
20   connect blowerCtrl.blwPower with hvacBlower.power;
21   ...
22 }
```

Listing 5.4: Excerpt of the HVAC System Specification

The system is composed of various component instances. A component might be instantiated multiple times like TemperatureAdapter and TemperatureFlapCtrl for the driver's and the passenger's seat. The component's ports are connected together. YAKINDU CoMo has built-in consistency checks for the component model. It checks for example whether all requiring ports are connected and whether requiring ports are only connected to providing ports and vice versa. Furthermore, the user is supported by context-sensitive auto-completion of keywords and identifiers.

5.5 Transformation into AMALTHEA Models

There is already a component model that is part of the whole AMALTHEA model. Nevertheless, in the HVAC demonstrator case study we used YAKINDU CoMo since it is much more expressive. The AMALTHEA components model is partly a one-to-one copy of YAKINDU CoMo but YAKINDU CoMo additionally includes amongst other things the declaration of a component's behavior and the linking from behavior declaration to its implementation by statecharts.

Though, further processing such as partitioning is purely based on the AMALTHEA model. The YAKINDU component model will therefore be automatically transformed into the AMALTHEA component and software model parts for further processing.

Figure 5.2 shows the AMALTHEA component model part after transformation from the YAKINDU CoMo. It is a nearly one-to-one transformation without the behavior specification. The AMALTHEA component model contains all the components and its ports defined in YAKINDU CoMo as well as the system composed of these components. Since the Franca IDL is already part of the AMALTHEA Tool Platform, the ports in the AMALTHEA component model are also linked to the interfaces defined in Franca IDL just like the YAKINDU CoMo component ports.

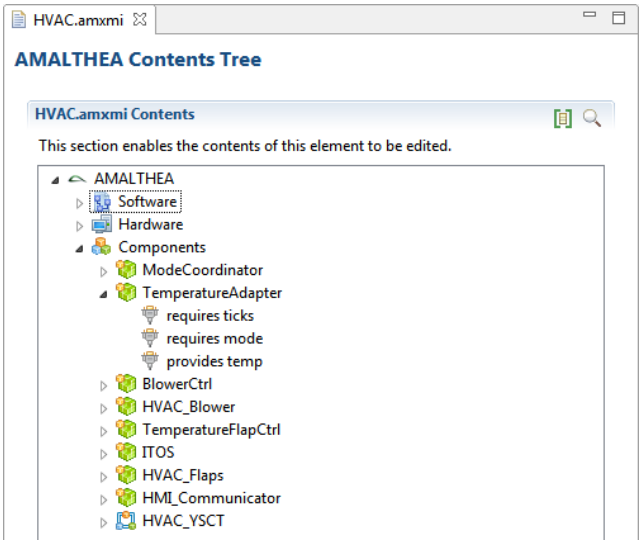

Figure 5.2: AMALTHEA Components Model

The behavior specifications of the YAKINDU component model are transformed into the AMALTHEA software model part. For each providing port, a label is added to the software model. Labels are means to exchange data between software components. The types of the labels are set according to the interface definitions. Figure 5.3 shows the labels section within the AMALTHEA software model.

A component's behavior describes an atomic unit that is usually executed periodically. Such an executable unit is called runnable. Several runnables are composed to a task to be run on the embedded target system. Thus, for each component instance in the system of Listing 5.4 and each of its behavior specifications a runnable is created in the AMALTHEA software model. Figure 5.4 shows a screenshot of the AMALTHEA model editor where the runnables are listed. To create a unique name for a runnable, the runnable's name is composed of the system's name, the component instance's name, and the behavior's name.

A behavior has a number of ports that it requires or provides. Since ports have been transformed into labels, each runnable has a list of labels that it reads (required port) or writes (provided port). Furthermore, each runnable has a property Execution Cycles that is created and set to 0 by the transformation. It describes the cycles a runnable needs to be executed once. The actual value of an Execution Cycle has to be set manually, either by estimating the value or by measuring it. The value is basis for further processing of the model in the partitioning phase where the runnables are composed to tasks.

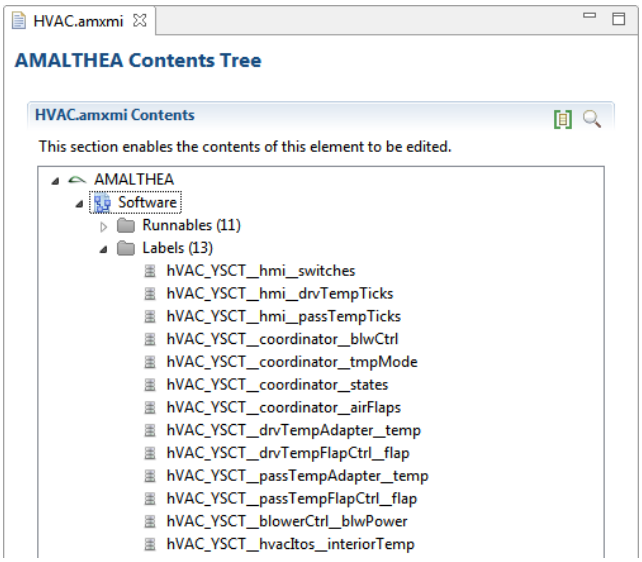

Figure 5.3: Labels in the AMALTHEA Software Model

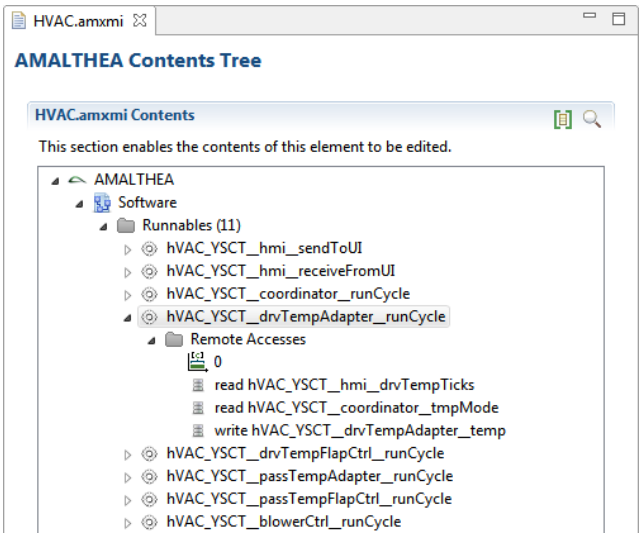

Figure 5.4: Runnables in the AMALTHEA Software Model

6 Partitioning

Partitioning in context of graph theoretical computing comes with a wide range of problems and methodologies, spread across a variety of applications. Besides most orientations, the partitioning in AMALTHEA rather focuses on directed acyclic graph (DAG) structures than on field structures or undirected vertices sets. Most importantly, cost functions define the algorithms and their result, aiming on different balanced sets, which shall be computed on separated computation units. The partitioning problems require efficient methods that are described and evaluated by analytical results in the course of this chapter.

6.1 Concept

Figure 6.1 shows an abstract view of the different features and also denotes the partitioning's idea, intention and necessity.

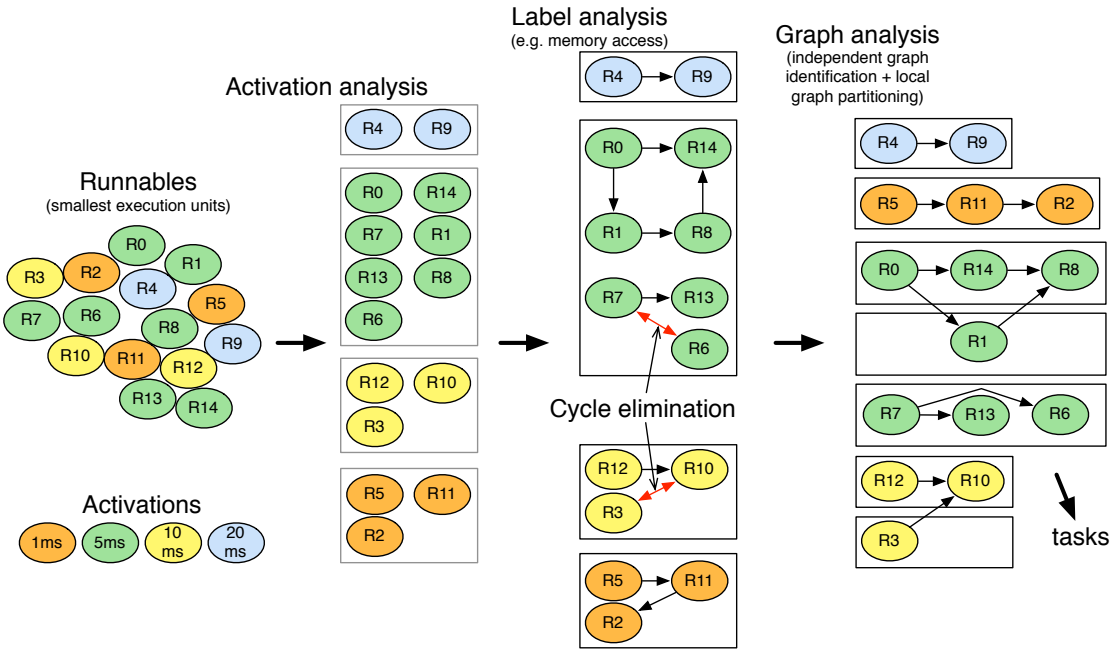

Figure 6.1: Partitioning concept

Figure 6.1 also depicts the in- and output of the partitioning shown in the left (input) and in the right (output) respectively. For illustration reasons, node weights are not

considered in the figure. The cycle elimination in the middle is a mandatory step, which has to be performed in advance of the independent graph partitioning and the local graph partitioning. Such cycle elimination also considers cycles among multiple runnables as stated in section 6.2.1. Moreover, the figure 6.1 shows all five features namely the Activation Analysis (AA), the Label access Analysis (LA), the Cycle Dissolution (CD, also denoted as cycle elimination), the Global Graph Partitioning (GGP) and the Local Graph Partitioning (LGP) partitioning approach. Adjacently to the graph partitioning, `ProcessPrototypes` are transformed to tasks and finally mapped to hardware specific processors. These two steps are described in section 7, as they further consider mapping and hardware constraints, which are part of the mapping process.

The partitioning approach has been integrated to the AMALTHEA Tool Platform and tested with a democar example model, which has been transformed from an industrial AUTOSAR model. Further examples for testing, problem definition and resolution are shown in each section correspondingly.

6.1.1 Terminology and Methodology

Before starting to introduce the partitioning concept and its technologies, this section gives a brief description of terminology and methodology in graph theory.

Graph theory plays important roles in computer science e.g. in program segmentation, sparse matrix reordering, floor planning, circuit placement, cluster ranging from computer vision, data analysis and other important disciplines. Directed acyclic graphs are thereby used in various application fields to model data dependencies (flows), system architecture, task constraints systems or similar approaches. One of the most commonly used methodologies on DAGs are topological algorithms in order to perform various functions e.g. sorting, graph position determination, source or sink identification and more. *Breadth-first-search* as well as *depth-first-search* algorithms are used with regard to topological orders for instance.

Forming computation sets, which is the partitioning's intent, mostly concerns the division of processes into subprocesses whereas each subprocess consists of computational load. In terms of graph theory these subprocesses are denoted as nodes. A node often reveals uni-directed communication with one or multiple other nodes, such that a directed edge between them denotes dependency as shown in figure 6.2.

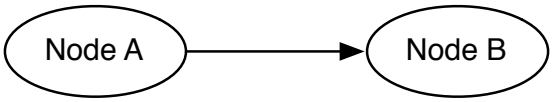

Figure 6.2: Node dependency

As seen in figure 6.2, `Node B` depends on a result of `Node A` and thereby depends on `Node A`. In case `Node B` is assigned to a different computation unit i.e. processor, the system must preserve the given ordering of both nodes. Otherwise `Node B` may be started without `Node A` being finished resulting in `Node B` termination violation. Such order may

be preserved via inter process activation concerning the process with `Node A` and explicit synchronization points at `Node A` (set) and `Node B` (wait).

As stated previously, the partitioning considers DAGs: $G = (V, E)$, whereas V defines the set of vertices (nodes / runnables, $\omega(v_i)$ = vertex i) and E defines the set of edges (dependencies / RunnableSequencingConstraints). Each vertex is defined by its computation cost (instruction cycles in terms of AMALTHEA) $\omega(v_i)$. Such graphs can be derived from task graphs as system-level descriptions. Edges in terms of AMALTHEA are denoted as RunnableSequencingConstraints. Any edge e possesses a parent v_p and a child v_c. However, `RunnableSequencingConstraints` provide further edge representation as stated in [2]. In case a node has no parent, it defines an *entry* node and a node, which has no child is denoted as an *exit* node. The partitioning process assumes communication cost between two nodes, which are assigned to the same task, to be zero. Communication cost is derived from the label size, which the runnables access and that defines the dependency between two runnables.

DAGs allow partial orderings to be derived from topological calculation such that node $A \subseteq$ node B iff $A \rightarrow B$, such that B depends on A. Furthermore, DAG structures benefit from significantly faster algorithms compared with algorithms for arbitrary graphs e.g. finding the shortest path calculates in linear time within DAGs ($O(|V| + |E|)$) [8] whereas the Dijkstra or the Bellman-ford algorithm for arbitrary graphs run in $O(|V^2|)$ [9] respectively $O(|V| \cdot |E|)$ [5] time.

In [8], several definitions are given upon graph data, which are used within the partitioning plugin:

- *span* = length of the critical path
- $work = \sum_{i=1}^{N} RT_{v_i}$ with RT_{v_i} = runnable's instructions
- work law: $T_P \geq \frac{T_1}{P}$, whereas T_P is the complete runtime on a system with P Processors and (T_1 = sequential runtime)
- span law: $T_P \geq T_\infty \rightarrow$ defines that the runtime of a system with P processors is always $\geq$ the runtime on a system with an unlimited number of processors
- $parallelism = \frac{T_1}{T_\infty}$
- $slackness = \frac{T_1}{PT_\infty}$ (factor by which the parallelism exceeds the number of processors in the system)

Amdahl's law defines that if a change improves a fraction f of the workload by a factor K the total speedup is [4]:

$$Speedup = \frac{Time_{before}}{Time_{after}} = \frac{1}{f/K + (1 - f)} \tag{6.1}$$

This equation corresponds to the *parallelism* factor given in [8].

It is important to note that the runtime of a partitioned system not only depends on the work and the span, but also on how many processors are available and how the

scheduler allocates partitions to processors. Furthermore, the more processors are used beyond the parallelism value, the less perfect the speedup is [8].

6.1.2 Partitioning Introduction

Partitioning influences system performance. The more efficient the partitioning process forms computation sets distributed among computation units i.e. processors, the more the systems benefits from meeting time restriction, energy demands or high performance real time applications. These aspects are common topics of interest in almost all areas of science and technology especially concerning the automotive research domain.

According to Foster [11], the partitioning step is intended to reveal parallel execution opportunities of a problem by partitioning it into fined-grained decompositions, providing the greatest flexibility for parallel algorithms. However, it should be avoided to replicate data or computations [1].

[14] lists a set of criteria which may be used to create tasks out of data-flow graphs:

- Functions which are dependent on a specific I/O shall be a separate task, i.e. not blocking other functions while waiting for input (addressed by the mapping process in section 7)
- High-Priority functions shall be separate tasks (addressed by the mapping process in section 7)
- Functions with a high amount of computations should be separate tasks with a low priority
- Closely related functions should be merged into one task to prevent unnecessary system overhead e.g. by context switches and/or data passing
- Periodic functions should be modeled as a separate task

Besides the fact that these guidelines were published in the mid-80s and focus on data-flow graphs, they still keep their significance and may be applicable on other types of software and behavioral models [1].

Most partitioning methods focus on equal computational loads across the partitions and minimized communication between these partitions. In terms of DAGs, the cut sets can be formed most likely focusing on two different methodologies. On the one hand a cut can be performed with horizontal orientation in order to form sets that compute sequentially across iterations and different iterations in parallel. On the other hand cuts with vertical focus can form sets that can be computed in parallel for one iteration and different iterations sequentially. Figure 6.3 exposes the two different cut methodologies and corresponding processor scheduling.

The small indices indicate the computation iteration due to DAGs define dependencies within a system, which are executed consistently over time i.e. computed over several iterations. The left i.e. horizontal cut method in figure 6.3 shows that different iterations are computed at the same time slice (column). This method provides a great balanced

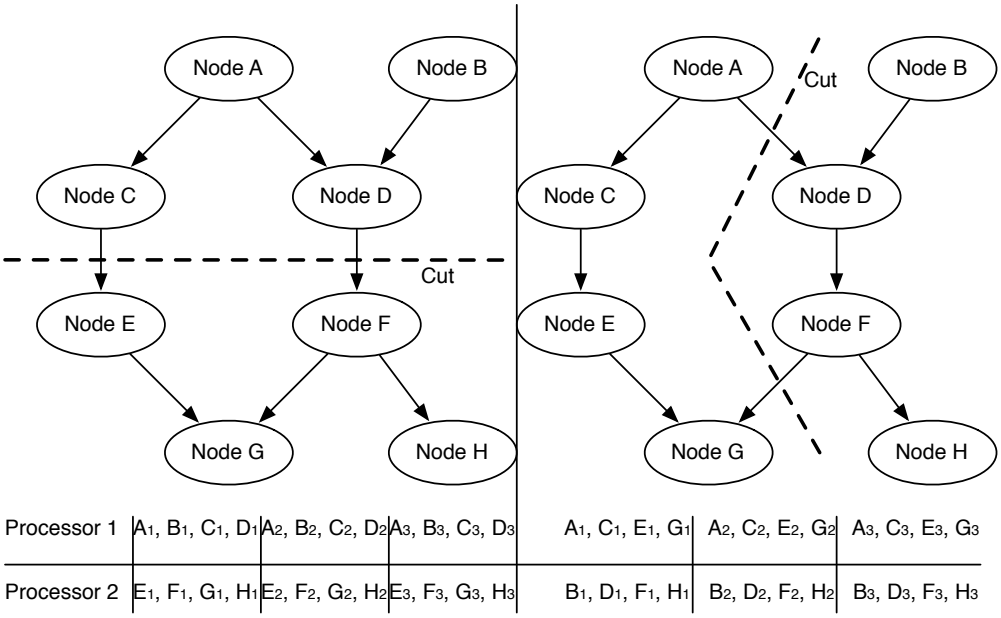

Figure 6.3: Two different DAG cut methods

load but a delayed complete iteration calculation. In contrast, the right i.e. vertical cut method provides early complete iteration calculation due to one iteration computation of the complete graph being calculated in parallel. Each iteration is calculated in a separated time slice (column). The vertical cut method is explored and evaluated within the AMALTHEA partitioning approach whereas horizontal cut methodologies can be investigated by further scheduling or mapping algorithms.

Besides ordering constraints, the following key performance overheads have to be considered:

- inter task control dependencies
- inter task data dependencies (register and memory)
- load imbalance
- serialization due to large tasks

Addressing these factors fundamentally affects the amount of exploited parallelism by choosing the tasks. In some cases such problems are addressed by the system's compiler due to their complex nature [18]. Nevertheless, the main cost functions for partitioning purposes are performance, power, footprint and scalability (through configuration).

6.1.3 Feature Introduction

Several features support the development of software for embedded multi-core systems in context with partitioning within the AMALTHEA platform by providing methods for

activation analysis, label access analysis, cycle elimination, independent graph identification, local graph partitioning and task number restriction based partitioning. These features are able to adapt a software model and a constraints model for the purpose of distributing software in an effective way, such that execution time and cross process communication is minimized and load is distributed evenly among partitions. For this purpose, runnable's activations (section 6.2.1) and label accesses (section 6.2.1) are analyzed in order to create `ProcessPrototypes` within the software model and `RunnableSequencingConstraints` within a new constraints model initially. At this development stage, the `ProcessPrototypes` are used to group runnables by activations and to facilitate `AccessPrecedences`. These `AccessPrecedences` are used to represent dependency on the one hand but to keep the dependency invisible to graph analysis on the other hand, in order to allow the edge's child node to work with older data for bypassing cycles. Implementations in early system design phases, which are not yet completed at that time, can be verified, structured and classified by semantic and purpose via such `ProcessPrototypes`. After the activation and label access analysis, directed acyclic weighted graphs (DAWGs) are build via information given the software and the constraints model, by using the JgraphT library. These graphs can be used in order to distribute software among a multi-core system via analyzing the call graph (partial orderings) and using algorithms to get the right order and parallelization of the elements and dependencies. This results in new `ProcessPrototypes` respectively a finer grained model, which features a first effective utilization of parallel resources i.e. a reduction of execution time.

Figure 6.4 depicts the configuration dependent file generation possibilities. Regarding

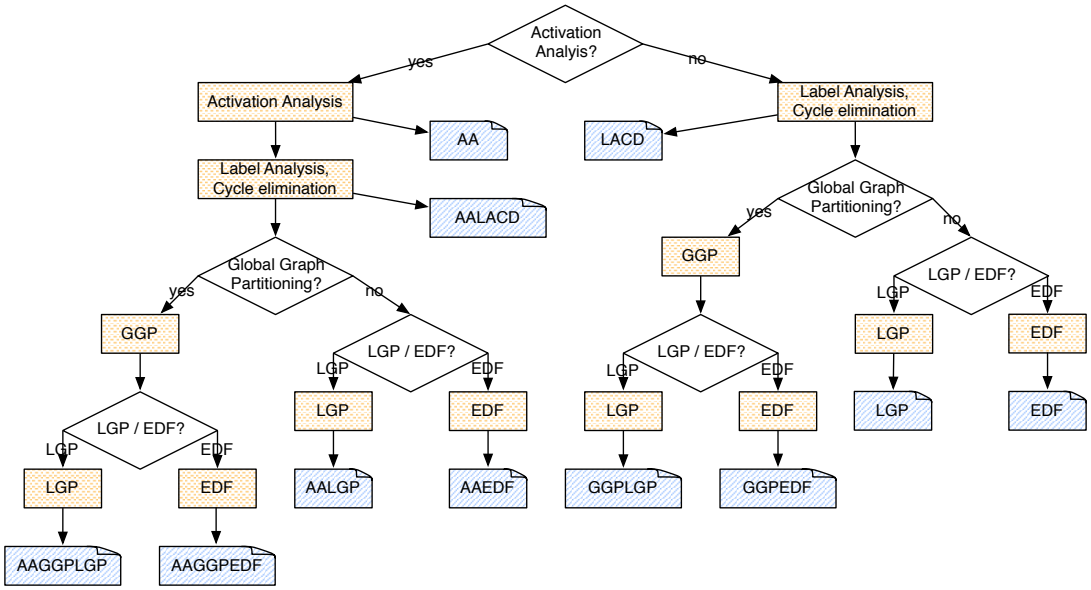

Figure 6.4: configuration dependent file generation flow chart

the democar example model ([12]) each *configuration path* generates a model, which features a different exploitation of parallelism.
Rectangles filled with diagonally lines (light blue) define files, which are generated, whereas rectangles filled with dashes (orange) define operations, which are performed within the path. The decisions among the graph, indicated by the rhombus objects, access the configuration parameters and thereby define which operation (denoted within orange rectangles) has to be performed. At the end, the user is able to configure the partitioning in eight different ways, whereas `LACD` or `AA` and `AALACD` files are mandatorily generated.

6.2 Implementation

6.2.1 Preliminary Features

In order to perform the actual partitioning on a DAWG (Directed Acyclic Weighted Graph), various adaptions need to be performed on the given input formed by a set of runnables with label accesses, instructions and activations. These adaptions create `ProcessPrototypes` for grouping runnables by their activations (section 6.2.1), a constraints model with `RunnableSequencingConstraints` representing runnable dependencies (section 6.2.1), `AccessPrecedences` for eliminating cycles within a graph (section 6.2.1) and `ProcessPrototypes` for identifying independent graphs (section 6.2.2).

Activation Analysis

Using industrial application abstractions, one realizes that runnables are given various activation parameters. Typical for embedded software, code fragments are executed in different intervals due to timer, interrupts or events. Sensors or actuators for example must often be read within short intervals for precise and accurate executions. Contrarily, certain processing fragments independent from the system's stability can be executed at greater intervals due to having less impact on the system's performance. Such activations can either be referenced by tasks (that correspond the `ProcessPrototypes`) via the stimulation model or by runnables via the software model. By assessing these model entities and references, temporal classification can be implied. These grouped runnables can be seen in figure 6.1, indicated by colored grouping (with respect to their activation).

Label Analysis

The next mandatory step in order to perform the partitioning is the analysis of each runnable's label accesses. Detailed information about labels is given in [2]. The label analysis comprises the comparison of read and write accesses of all runnables for identifying dependencies. For example, in case runnable A writes a label and another runnable B reads the same label, runnable B depends on runnable A. This dependency is saved as a `RunnableSequencingConstraint`. A `RunnableSequencingConstraint` is the basis for the DAG analysis and allows the determination of concurrency due to giving information

about fork and join dependencies respectively runnables that can be calculated in parallel. Furthermore, the label analysis allows deriving memory structure via common data accesses. The `RunnableSequencingConstraints` are stored within a new or an already existing constraints model. The structure of a `RunnableSequencingConstraint` is given in figure 6.5.

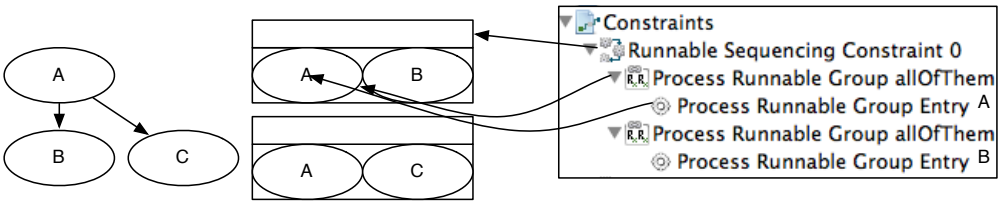

Figure 6.5: RunnableSequencingConstraints

Figure 6.5 shows a graph, consisting of three runnables on the left side and the corresponding `RunnableSequencingConstraint` constructs in the middle. For reasons of simplicity, each dependency of a graph is modeled as a separate `RunnableSequencingConstraint`. Figure 6.5 also shows the corresponding model within the AMALTHEA platform as a screenshot on the right side. Such a constraint possesses two groups, with the source runnable in the first group and the target runnable in the second group. At this point of development, the `RunnableSequencingConstraints` do not feature any process scope (since no `ProcessPrototypes` are present at this time) nor other parameters except the *RunnableGroupEntry* parameter "all of them", indicating, that all of the runnable group's entries have to be called.

Cycle Elimination

The cycle elimination is a mandatory step for all subsequent methods and features. Topological and graph theoretical calculations require DAWGs, such that a cycle elimination has to be performed in advance. The AMALTHEA partitioning approach features directed graph cycle elimination via converting `RunnableSequencingConstraints` into `AccessPrecedences`. This is used in order to retain a dependency and to tell later development phases, that the `AccessPrecedence's` child node shall execute with values from previous calculation iterations. A cycle may occur in case two runnables share the same resource (label) in both directions, i.e. both runnables read and write the same label, or in case runnable A reads label BA and writes label AB and runnable B reads label AB and writes label BA. Furthermore, a cycle may be formed across multiple runnables. For the purpose of finding such cycles the `JGraphT` -library is used. After all cycles have been identified, a specific mechanism detects edges (dependencies), which occur in multiple cycles. This mechanism iterates over edges within mutliple cycles descendingly, i.e. it starts with an edge, which occurs in most cycles for ensuring minimal edge elimination. In order to retain a dependency that has been selected for elimination, such edges are transformed from a `RunnableSequencingConstraint` to an `AccessPrecedence`. Usually, in industrial development, such `AccessPrecedences` are given from the OEM, respectively

the software developer, who is responsible for transmitting cycle free software. Hence, this manual `AccessPrecedence` generation takes a lot of time and effort since there is no explicit automation. This is where AMALTHEA highly benefits from the partitioning plugin's cycle elimination. The current cycle elimination implementation works great with smaller models without much effort. However, bigger models, featuring more than a thousand runnables, often reveal structures, which take a lot of time for decomposition. Various code optimizations shall be addressed in further work for industrial applicability.

After edge sharing cycles have been decomposed, all cycles, which do not share any edges, have to be decomposed as well. For each of these cycles, an edge is identified, that provides an optimal retaining graph. Figure 6.6 outlines, how different edge decompositions affect the resulting graph.

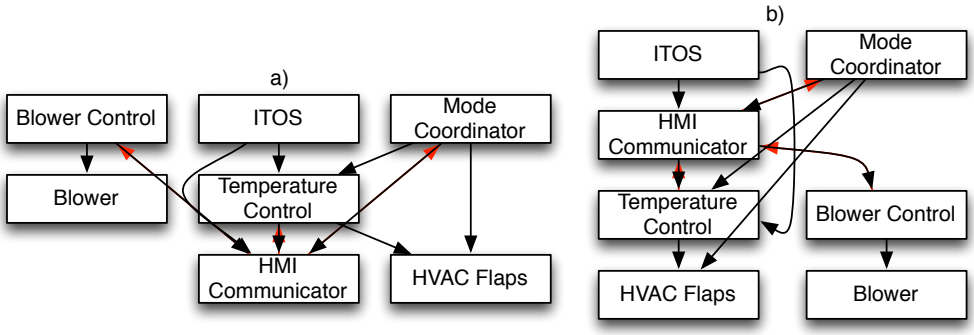

Figure 6.6: Efficient graph forming within cycle elimination

Figure 6.6 exposes two possible different cycle elimination results of the HVAC model at a specific development step. At that point of development, the HVAC model features three bilateral communications, which constitute the lowest level of cycles. Red transitions indicate edges, which are decomposed into `AccessPrecedences` for the corresponding solution. For illustration purposes, we assume equal execution time for each runnable. The actual number of cycle elimination possibilities is $2^3 = 8$, whereas 6 solutions are valid and two out of these six are shown in the figure 6.6 as *a) and b)*. *a)* features a best minimal runtime for two tasks and *b)*features even a lower runtime for three tasks. This assessment is made with respect to topological graph structure respectively the *span* of a graph (Critical Path) compared with its *parallelism*.

6.2.2 Independent Graph Identification

The independent graph identification process, or sometimes also denoted as global graph partitioning, can be executed after the cycle elimination in case the user selected the corresponding entry in the configurations. This methodology looks for graphs or single nodes, which do not access any label that other runnabels or graphs access as well. Such methodology allows forming tasks, which can be totally distributed to either different cores or even to totally different systems. Furthermore, such independent sections

can also be transformed into components, providing modularity and reusability. The Democar example (see [12]) for instance features four independent graphs.

6.2.3 Local Graph Partitioning (LGP)

The LGP approach considers node weights (i.e. arbitrary computation) and partitions DAWGs, whereas loads (weighted nodes) are equally distributed among an automatically determined number of partitions (the predefined number of partitions approach is described in section 6.2.4). The partitioning's objective is to reduce execution time and inter task communication. The subsequent mapping methodology [16] further considers resource constraints (availability of hardware and software resources namely program and data memory).

As the LGP approach initially computes the critical path of a DAG, the following definitions are outlined:

DEFINITION 1

A critical path of a task graph is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation costs and communication costs is the maximum. [17]

DEFINITION 2

A critical path is a longest path through a DAG, corresponding to the longest time to perform any sequence of jobs. Thus, the weight of a critical path provides a lower bound on the total time to perform all the jobs. [8]

Mentioning both definitions here is important, because definition 1 explicitly expresses the communication costs whereas definition 2 emphasizes on the fact, that the critical path serves as the lower bound on the total graph's computation time.
According to the partitioning process, this critical path is assigned to the first partition and branches of the graph are subsequently assigned to additional partitions. The approach has been chosen, because the critical path features mandatory sequential ordering, that can not be computed in parallel. However, another partitioning approach could be possible, that does not feature the critical path, but horizontal cuts instead, as stated in figure 6.3 section 6.1.2. The following pseudo code illustrates the local graph partitioning algorithm:

```
1  Sort Nodes topologically and gain information about earliest initial time (eit) and
       latest initial time (lit)
2  Let T denote the set of tasks
3  Determine the graph's critical path CP and assign it to the first task t in T
4  Let U denote all unassigned nodes
5  WHILE U is not empty
6    create task t_x, set tt=0;
7    WHILE taskTime tt < CPTime
8      let an denote the set of assignable nodes according to tt and each node's
           timeframe
9      SWITCH an.size
10       CASE 0:
11         No assignable node → increase tt to eit of the next applicable node
12       CASE 1:
13         Assign the node (an[0]) to t and remove it from U
14       CASE >1:
```

```
15          let co denote the set of communication overheads of each node in an
16          let ts denote the set of timeslices, that each node is applicable to
17          determine the most effective node men in an, thats co value is small and
                ts is most restricted according to eit and lit
18          assign men to t, increase tt correspondingly and remove men from U
19      ENDSWITCH
20    ENDWHILE
21  ENDWHILE
```

Listing 6.1: Pseudocode for local graph partitioning algorithm

The algorithm shown in listing 6.1 iterates among unassigned nodes (line 5) and time slots (line 7). For each iteration, nodes (runnables) are identified, that can be assigned to the specific time slot according to the ordering constraints (line 8). Afterwards, in lines 9 to 18, either the time slot is increased in case no node is assignable (lines 10-11), the only assignable node is assigned to the task (lines 12-13), or one node out of the previously calculated set of nodes is identified as the most effective node and assigned to the current task (lines 14-18). In the latter case, the determination of the most effective assignable node is done with regard to the node's communication overhead (line 15) and the time frame each nodes is assignable to (line 16).
In order to comprehend the node's time frame consideration process, an example shall give a better understanding. We assume that at time slot *t=1* nodes *A, B,* and *C* are assignable. The node's time frames are: $TF_A = \{1-4\}, TF_B = \{1-3\}, TF_C = \{1-1\}$ and the weights are $W_A = 5, W_B = 1, W_C = 2$. Due to each node is assignable to task time 1, the algorithm selects node C initially, because its time frame is most restricted (line 16-17). Afterwards, at task time 3, nodes A and B are still applicable and node B is chosen again because of its time frame restriction. The result would be $C - BA - - - -$ (whereas the dashes indicate the previous node's execution).
In order to get the node's time frames, each node (runnable) is given an earliest initial time `eit` and a latest initial time `lit` (line 1 in listing 6.1). At this point it is important to mention, that `lit` requires a critical path, whereas `eit` does not. This requirement is caused by the fact that the latest initial time refers to the critical path and can not be computed without it. The necessity of a critical path for the determination of a node's latest initial time respectively the calculation of $G4_{lit}$, requires the critical path time as shown in equation 6.3.

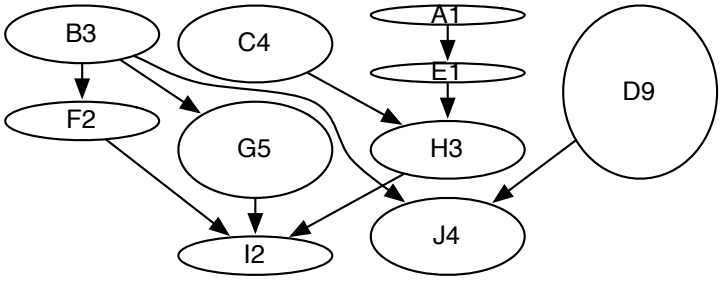

Figure 6.7: DAWG with 9 nodes

Figure 6.7 exposes the graph $G = \{A1, B2, C4, D9, E1, F2, G5, H3, I2, J4\}$ with $CP = \subset$

$G = \{D9, J4\}$ whereas the values after each notations also describes the node's weight i.e. $B3$ features weight 3. The weight of a node is also indicated by the node's height. For instance we want to calculate $F2$'s `eit` and `lit` values. `Eit` is calculated with regard to $B3$ only, that features a runtime of 3. So $B3$ consumes slots $0-2$ such that $F2$ can be started at the earliest slot of 3 due to the just mentioned runtime of $B3$. In other words, with $LPPRT()$ describing the *longest parent path runtime* we get:

$$F2_{eit} = LPPRT(F_2) = \sum_{i=1}^{n} RT(v_{LPP_i}) = RT_{B3} = 3 \tag{6.2}$$

Since vertex $F2$ just features $I2$ as a child vertex, its *lit* value is calculated via

$$\begin{aligned} F2_{lit} &= CPtime - LSPRT(F2) - RT_{F2} = CPtime - \sum_{i=1}^{n} RT(v_{LSP_i}) - RT_{F2} \\ &= 13 - 2 - 2 = 9 \end{aligned} \tag{6.3}$$

whereas $LSPRT()$ defines the longest succeeding path runtime. With these values, we can say, that *F2* can be started between time slots 3 and 9. By means of these values, the algorithm determines which set of vertices shall be assigned to additional tasks in what order. Having the algorithm performed on the graph shown in figure 6.7, the result looks like figure 6.8.

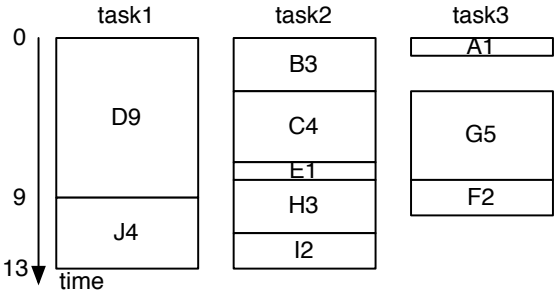

Figure 6.8: LGP Gantt chart result on example DAWG 6.7

Three tasks are shown in figure 6.8 horizontally, whereas time proceeds vertically top down. The LGP approach features minimal runtime due to critical path consideration, but also a possible high amount of partitions. Depending on the graph structure, the LGP always determines the number of tasks automatically, so that a task number restriction is not possible. However, for task number restriction, the following section 6.2.4 describes an implemented alternative to the LGP approach.
To sum up, the local graph partitioning approach features a theoretically optimal load balancing among automatically generated partitions, which can be computed in parallel with respect to ordering constraints. The optimal parallelism is achieved through a unbounded number of tasks within this approach.

6.2.4 Earliest Deadline First Partitioning

The EDF partitioning was developed for allowing the user to restrict the number of tasks. This may be important and useful in very large systems in order to keep the task creation and task inter communication costs low. The EDF partitioning balances runnables across partitions with respect to their logical orders expressed by the following pseudo code:

```
1  Let R denote the set of all runnables
2  Let T denote the set of tasks (+init)
3  Let A denote the set of assigned nodes
4  WHILE A.size < R.size
5    let edr denote the runnable with lowest eit
6    let tdi denote the indices of tasks, that edr is dependent of
7    SWITCH (tdi)
8      CASE 0:
9        assign edr to task, that features the lowest tt / utilization
10     CASE 1:
11       assign edr to task, that edr is dependent to
12     CASE >1:
13       assign edr to task with latest dependency
14   ENDSWITCH
15 ENDWHILE
```

Listing 6.2: Pseudocode for partitioning algorithm

The pseudo code in listing 6.2 exposes the mechanism of assigning a set of runnables to a given number of tasks with respect to their logical orders. A specific feature of the implementation is the consideration of runnables' time frames. The time frame calculation is already expressed in section 6.2.3. This has been developed in order to not randomly distribute the runnables, but taking orderings and inter communications into account in a way, that the load is distributed most evenly. Line 5 determines the (earliest) unassigned node. Line 6 expresses the task dependency determination. Therefore, all current node's parents are compared with each latest assigned node at each task. In case a parent equals a last task's node, lines 7-14 define to which task the node is assigned to. For illustration purpose, the example in figure 6.7 shall be EDF partitioned into two partitions featuring as even loads as possible. Working through the pseudo code with this example, the algorithm starts with assigning (*edr* =) *A1* to the first task, as it features the earliest initial time (eit_A) of *0* (lines 8 and 11 in listing 6.2). However nodes *B3, C4* and *D9* feature the same *eit*, but are not selected due to their longer runtime. Afterwards *B3* is detected in line 8 (eit_B=0) and assigned to the second task. The next two iterations assign *C4* and *D9* to tasks one and two again via line 11 in listing 6.2. The fifth iteration detects *E1* as the node with the lowest `eit` = 1. Its parent (*A1*) it not located at the last time frame at any task, so *E1* is assigned to task one, featuring the lower utilization (5 instead of 12 at task two). Iterations 6 to 10 assign *F2* to *I2* to the task with lowest utilization correspondingly due to the same reason compared with *E1*. The final result is shown in figure 6.9.
However using the EDF partitioning approach for defining three tasks the result is shown in figure 6.10.
Figure 6.10 denotes a special case as node *I2* is assigned to task two instead of task three, since it provides dependencies to nodes *F2* and *H3* at task two but only to *G5* at task three. This fact will be executed by line 13 in listing 6.2. Comparing this result with the

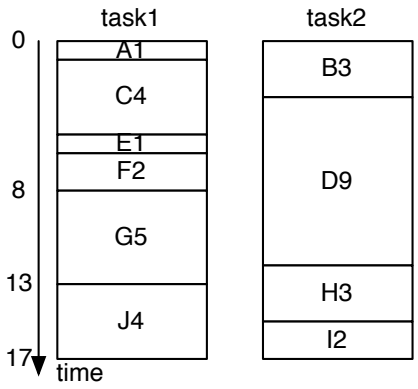

Figure 6.9: EDF 2 Gantt chart result

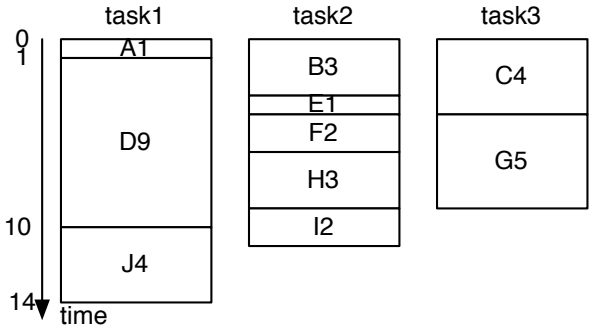

Figure 6.10: EDF 3 Gantt chart result

LGP result shown in figure 6.8, an increased overall runtime of one slot (by task one) but though less inter communication can be realized.

A special case occurs, when the activation analysis created two `ProcessPrototypes` for instance, and the user wants three tasks to be created. In that case, the EDF partitioning approach is able to determine, which of the two existing `ProcessPrototypes` features more work or more parallelization potential and selects the activation Prototype for EDF partitioning correspondingly.

To sum up, the EDF partitioning approach features an effective load balancing among a given number of tasks with respect to node's ordering constraints.

6.2.5 Other Features

Besides the features described in the previous sections, the partitioning plugin provides further features in order to enable visualization and configuration to the user.

Configuration

Figure 6.11 shows the partitioning's configuration panel within the AMALTHEA platform.

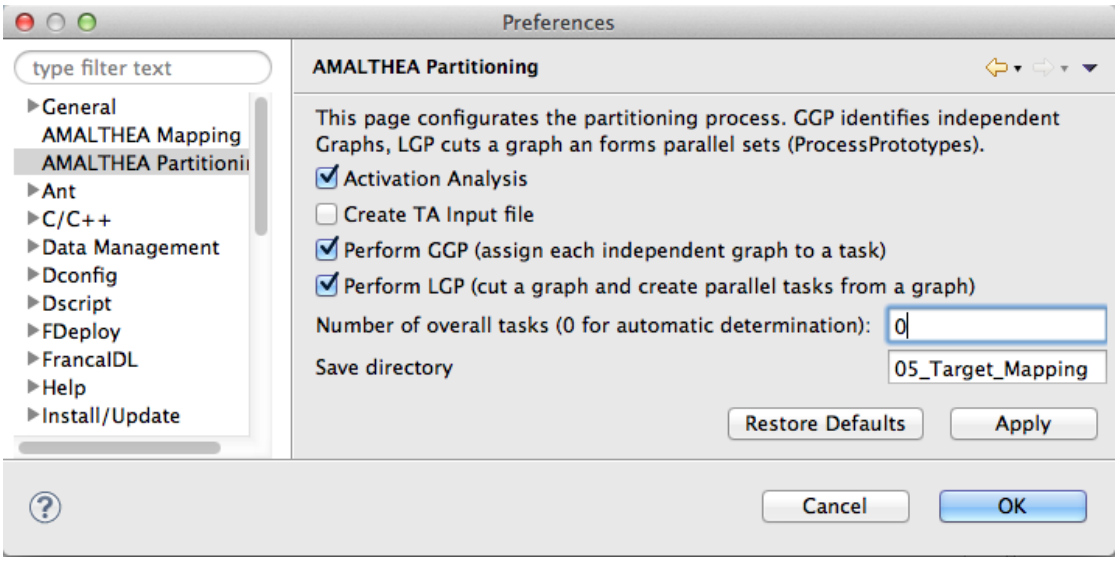

Figure 6.11: AMALTHEA partitioning configuration panel

The partitioning's configuration panel shown in figure 6.11 allows the user to select the different previously described features in order to generate files according to paths in figure 6.4. Two editor fields provide the definition of the number of tasks (only integer values are valid, that is verified during entering) that automatically enables the EDF partitioning approach (LGP is performed if this field is 0) on the one hand, and the definition of the output folder on the other hand. For future work, this configuration shall be combined with other AMALTHEA specific configurations, such as the mapping or the tracing configurations.

Applet Generation

The Applet generation is a specific command, which calls `writeApplet.java` instead of `partitioningManager.java` class. This class uses simple *buffered writer* object in order to generate a java class, which can be started as an applet with the help of the *JgraphT* library. For this purpose, two methods append static text to the generated class and other methods append dynamic text according to the read software and constraints model. These dynamic parts address JgraphT specific method calls, such as `addVertex(name)`, `addEdge(name)` and `positionVertexAt(Vertex, x,y)`. The last part (`positionVertexAt(Vertex, x,y)`) features a specific mechanism, since x and y coordinates have to be generated in a way, that the applet provides a reasonable visualization. For that purpose, the generation uses global counters in combination with multipliers, which adjust their value with regard to the node's topological sorting.

The basic benefit of an applet, compared with a Graphviz file, is the ability to manually edit the visualization within the applet viewer. Graphviz does not provide such editing within the viewer.

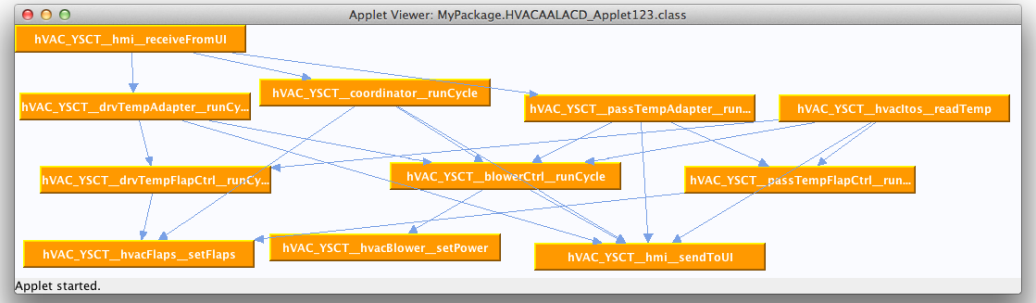

Figure 6.12: Applet visualization

An example Applet can be seen in figure 6.12 that expresses the HVAC demonstrator as described in [**?**].

TA Input Generation

This TA input file generation feature focuses on a constraints model approach, which features a different dependency representation compared with the result from the label analysis from section 6.2.1, that always features two *RunnableGroups* with each one *RunnableGroupEntry* entity within a RunnableSequencingConstraint. The approach can be adapted to feature more *RunnableGroups* and more *RunnableGroupEntries* and a less amount of RunnableSequencingConstraints, derived from the same graph. An example is expressed in figure 6.13.

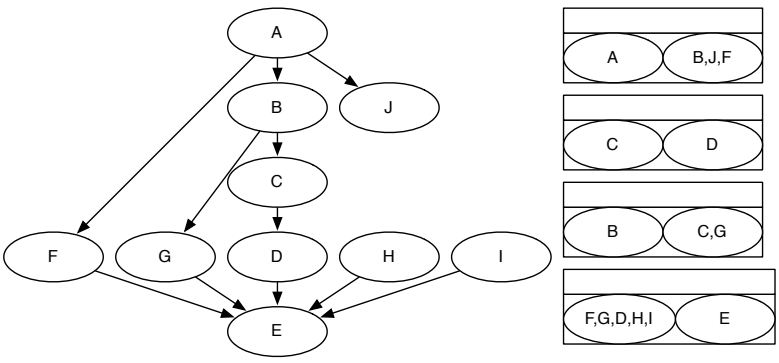

Figure 6.13: TA flexible RSC generation

Whereas the current implementation features a separate `RunnableSequencingConstraint`

for each arrow (edge / dependency) of figure's 6.13 graph (11 `RunnableSequencingConstraints`), the `RunnableSequencingConstraints` above the graph (4 `RunnableSequencingConstraints`) reveal the more compact approach. The generation has already been successfully implemented, although the reuse of such an approach requires various mechanisms and adaptions in order to cover more flexibility according to distributing the nodes. Since the nodes in the group must be executed in the order of the group, regarding the top left `RunnableSequencingConstraint` in figure 6.13, node *A* has to be executed before *B,J,* and *F*. The *runnable-order-type* property and the *process scope* reference can be set accordingly. The order types are already discussed in [2]. With the help of this generation the graph structure can be analyzed and optimized using industrial optimization tools like the TA toolsuite.

Source Model	Element	Description
SW	`Runnable`	`Runnables` will be distributed among `ProcessPrototypes` based on their `Instructions` and `LabelAccess` attributes
	`Activation`	The activation specifies the recurrence of the `Runnable`. The lowest recurrence is used to specify the overall deadline of all `Runnables`, i.e. the max amount of time for the sum of all `Runnable` executions
	`Process-Prototype`	Desfines raw data of a task. Is used for partitioning process, refers activations and features `TaskRunnableCalls` referring runnables.
	`Label`	Define values or memory structures, which runnables can access. Accesses to `Labels` are used for `Runnable-Sequencing-Constraint` generation
Constraints	`Process-Runnable-Group` `Runnable-Sequencing-Constraint`	Are used to determine the executional order of the `Runnables` as well as their interdependencies
Stimulation	`Activation`	Maybe referenced by Tasks. Used for grouping Runnables

Table 6.1: Processed information by partitioning strategy for minimizing execution time

7 Mapping

One of the essential steps during embedded software development for multi-core platforms is the mapping from elements of the software to elements of the hardware, e.g. `Task`s to `Scheduler`s, `Label`s to `Memory`s etc. This is usually a non trivial task, as an infinite number of combinations arises if either the software or the hardware becomes very complex. The purpose of AMALTHEAs Mapping Plugin is to determine such a mapping and store it in a ***Mapping Model*** which will contain the allocations of elements of the ***Software Model*** to elements of the ***Hardware Model***.

This is chapter is structured as followed: The first section provides an overview about the concept behind the mapping plugin. Section 7.2 contains details about the implementation and its algorithms. Last but not least, instructions how to utilize the plugin, as well as a detailed walkthrough based on AMALTHEAs HVAC example, are given in Section 7.3.

7.1 Concept

The conceptual implementation of AMALTHEAs Mapping Plugin is shown in Figure 7.1. As shown in the top of this figure, it requires several models to operate. The models for **Software**, **Hardware** and **Constraints** are mandatory while the **Property Constraints** Model is optional.

Using AMALTHEAs Mapping Plugin, the user is able to choose between different mapping strategies. Currently these strategies are split into two categories: *Heuristic methods* and *Integer Linear Programming (ILP) based methods*. Unlike ILP based methods, Heuristic methods, such as the *Heuristic Data Flow Graph (DFG) load balancing*, will immediately create a mapping.

ILP based methods on the other hand will first 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*. Once the ILP model has been created, it will be solved by one of the mathematical *Solvers*. Currently, the open source project Oj!Algo[1] has been used in AMALTHEAs Mapping Plugin. Furthermore, the user can activate an optional MPS generator, which will generate an MPS file containing the ILP problem. This file may be used to solve the ILP problem by external (e.g. commercial) solvers, which tend to be more efficient in solving larger models compared to open source Java implementations.

Once a mapping has been determined, it is displayed within the eclipse console and following output models are generated:

[1] Oj!Algorithms, licensed under the MIT license, see: http://ojalgo.org

- **Mapping Model**, containing the allocations from `Tasks` to `Schedulers`
- **OS Model**, containing the `Schedulers` for each `Core`
- **Stimuli Model**, containing the `Stimuli` for the resp. `Runnable` and `Task` activations

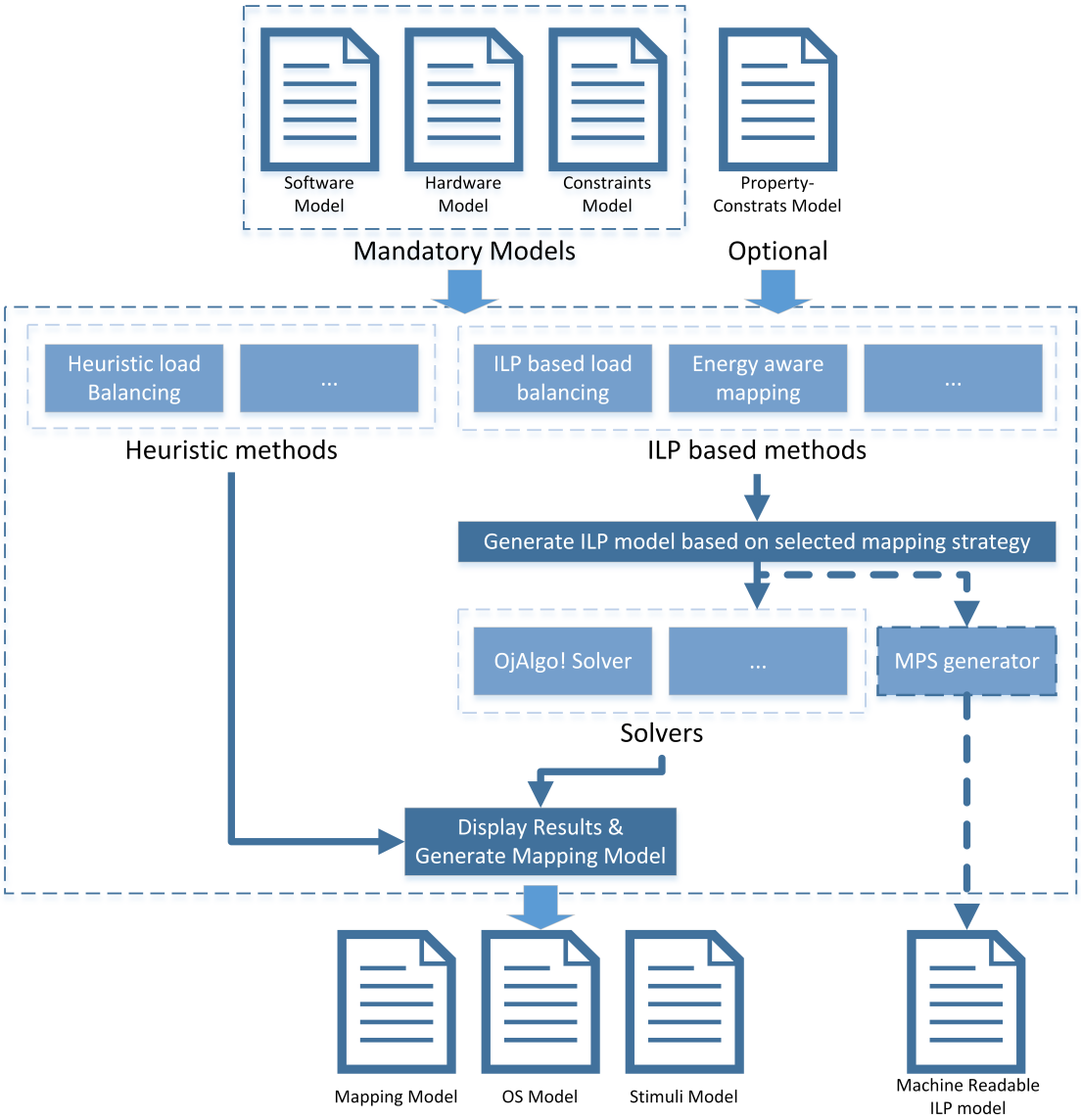

Figure 7.1: Concept of the AMALTHEAs Mapping Plugin

7.2 Implementation

The following subsections give a short introduction about the different algorithm implementations of AMALTHEAs Mapping Plugin. Section 7.2.1 describes the task generation method which is used to convert process prototypes into tasks. It is meant to be used by mapping algorithms which do not feature task generation by themselves. Sections 7.2.2 and 7.2.3 describe a heuristic and a mathematical load balancing approach for mapping of tasks to cores. Last but not least, a more complex method for energy efficient task mapping with its own task creation algorithm is outlined in section 7.2.4.

7.2.1 Task generation

The task generation method in AMALTHEAs Mapping Plugin is a pragmatic way to create tasks for other mapping algorithms which require `Task`s, i.e. are not designed to agglomerate `Runnable`s into `Task`s on their own. This step utilizes `ProcessPrototype`s which are generated by the partitioning plugin (see Chapter 6) and transforms them into `Task`s. Furthermore, it will also create the ***Stimuli Model*** which contains the activation elements for the `Task`s, i.e. `Periodic`. An overview about the transformed elements and their sources as well as destinations is shown in Table 7.1.

Source Model	Source Element	Target Model	Target Element
SW	ProcessPrototype	SW	Task
SW	Activation	Stimuli	Stimulus
Constraints	ProcessRunnableGroupEntry	SW	TaskRunnableCall

Table 7.1: Transformations performed by task generation algorithm

7.2.2 Mapping Strategy 1: Heuristic DFG load balancing

The *Heuristic Data Flow Graph (DFG) load balancing* algorithm aims at achieving an equal utilization of a hardware platforms cores for DFG based software models.

The first step in this algorithm is to determine the most complex `Task` (usually representing the critical path) and allocate it to the best fit core of a hardware platform. The runtime for each `Task` will now be estimated for every `Core` within the `System` and allocated to a `Core` which has the smallest increase of the longest overall runtime within all cores.

One of the major benefits of this algorithm is its very low runtime. The information which is processed by this mapping strategy and, as such, has to be present in the input models, is shown in table 7.2.

7.2.3 Mapping Strategy 2: ILP based load balancing

A comparatively simple ILP based strategy to allocate tasks to processors while minimizing the total execution time is presented in [10]. This method supports multiple processors with the same processing speed (e.g. homogeneous processors) and it does not consider any dependencies between the tasks (e.g. waiting for the results of the predecessor).

As described by the author, load balancing within this method is achieved by minimizing the highest execution duration C_{max} of all m processing units with n tasks (see eq. 7.1 and 7.2). The variable x_{ij} is set to 1 if a task j is allocated to processor i and 0 otherwise. Eq. 7.3 guarantees that each task is allocated to exactly one processor while 7.4 limits variables x_{ij} type to boolean values. The duration of a task j is specified by p_j.

$$\text{minimize} \quad C_{max} \tag{7.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} x_{ij} p_j \leq C_{max},\ i = 1, \ldots, n \tag{7.2}$$

$$\sum_{i=1}^{m} x_{ij} = 1,\ j = 1, \ldots, n \tag{7.3}$$

$$x_{ij} \in 0, 1 \tag{7.4}$$

One of the downsides in this algorithm is caused by p_j in eq. 7.2 which forces an equal processing duration of a task j on all cores. It is however possible to expand the method to support heterogeneous processors (in this case: processors with different processing speeds) with a minor modification[2]: replacing p_j with p_{ij}, i.e. a separate processing duration of task j for every core i, will solve this problem.

The minimal amount of information which is required to execute this algorithm is outlined in Table 7.2.

7.2.4 Mapping Strategy 3: Minimizing Energy Consumption

This mapping algorithm is based on the work *"Task Scheduling and Voltage Selection for Energy Minimization"* from Zhang et al. which presents a framework which aims at minimizing the energy consumption of variable voltage processors executing real time dependent tasks. This method is implemented as a two phase approach which integrates

- **Task assignment**: allocating each task to a core
- **Task ordering**: ordering the tasks in due consideration of their constraints and deadlines
- **Voltage selection**: selecting a slower but less energy consuming processor mode in order to save energy without harming any constraints or deadlines

[2]Mentioned, among others, in [15]

Source Model	Element	Description
HW	`Core`	A `Core` represents the target of an allocation, an OS Model with a `Scheduler` for each `Core` will be generated.

	`CoreType`
	`Prescaler`
	`Quartz`
		A `Core`s `Prescaler`, the referenced `Quartz` and the `CoreType`s attribute `CyclesPerTick` of a `Core` are used to determine the number of processed `Instructions` per second.

SW	`Task`	`Task`s will be allocated to a `Core` (over the `Core`s `Scheduler`)
	Runnable	`Runnable`s are derived from a `Task`s `TaskRunnableCalls`, their attribute `Instructions` is used during the load calculation for each `Core`
Stimuli	`Stimulus` (`Periodic`)	The `Periodic` Stimulus is used to specify the `Task`s activation rate, i.e. the period between its calls

Table 7.2: Processed information by heuristic DFG and ILP load balancing

In the first phase, opportunities for energy minimization are revealed by ordering real-time dependent tasks and assigning them to processors on the respective target platform.

- On **single processor platforms**, the ordering of tasks is performed by applying an Earliest Deadline First (EDF) scheduling. An further allocation of tasks to processors becomes needless, as only one allocation target exists.
- On **multi processor platforms**, a priority based task ordering is performed. The allocation of tasks to processors is determined by a best fit processor assignment.

Once the scheduling is created, there will be time frames between the end of one task and the start of another during which the processor is not being utilized (so called slacks). These slacks the prerequisites for the second phase, which performs the voltage selection. This phase aims at determining the resp. (optimal) processor voltage for each of its task executions without harming the constraints and eventually minimizing the total energy consumption of the system. In order to determine these voltages, the task scheduling is transformed into a directed acyclic graph (DAG) that is used to model the selection problem as integer programming (IP) problem. Once the model has been set up, it is optimized by a mathematical solver.

This algorithm has been implemented with two constraints:

- Only multi-core systems may be chosen as target platform, single core is not supported.
- The algorithm is not meant to be used on heterogeneous cores, i.e. only multiple instances of the same type of core are supported.

Table 7.3 lists the minimal amount of information which has to be present in the input models in order for this mapping strategy to work as well the special annotations which are added to the mapping model.

7.3 Utilization of the AMALTHEAs Mapping Plugin

This section provides information on the utilization of the AMALTHEA Mapping Plugin, i.e. its configuration (section 7.3.1) and how to generate mappings (section 7.3.2).

7.3.1 Configuration and Preferences

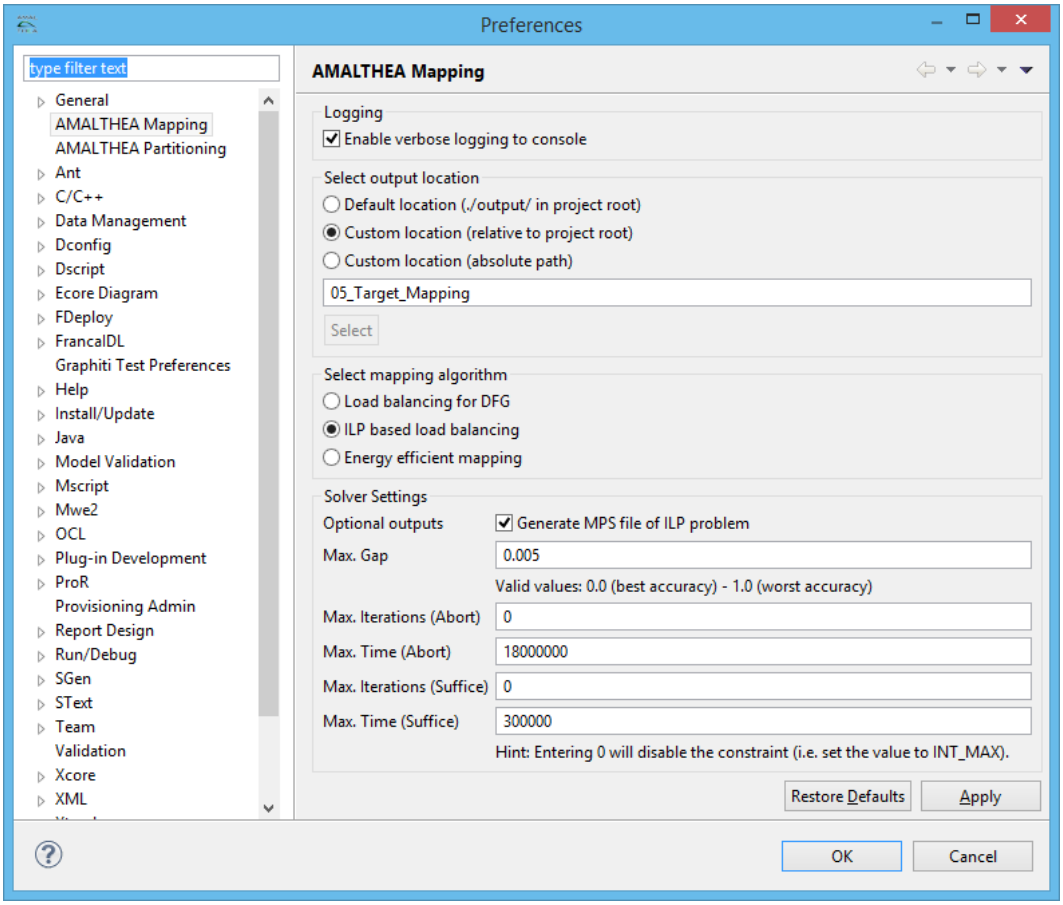

Figure 7.2: AMALTHEA Mapping Plugin's Preferences Page

The configuration of AMALTHEAs Mapping Plugin can be performed through its preferences page (see Figure 7.2). It is integrated into the AMALTHEA Tool Platform and can be accessed through the menu bar under 'Window' → 'Preferences' → 'AMALTHEA Mapping'. The configurable fields, their types and their descriptions are listed below.

Enabling verbose logging

Checking the box ***'Enable verbose logging to console'*** will enable verbose logging to stdout. This may help to identify problems if the mapping plugin should fail to generate a mapping.

Specifying the output location

The radio buttons under ***'Select output location'*** allow to customize the directory which where newly generated files will be placed into.

- **Default location** specifies the 'output' directory within the respective projects root folder as destination for all new generated files.
- **Custom location (relative to project root)** will ensure that all new files are created in a custom directory of the respective projects root folder. The name of this custom directory can be customized by changing the value in the text field below the radio buttons.
- **Custom location (absolute path)** allows the user to specify a full custom destination for new generated files. The destination can be specified by either changing the value of the text field below the radio buttons, or by using the "Select" button.

  Hint: It should be noted, that using this option will NOT update the project explorers folder list once the mapping is finished. It should be avoided to use this option in combination with a target location within the eclipse workspace.

Selecting a mapping algorithm

The radio buttons within ***'Select mapping algorithm'*** allow to customize the mapping strategy which should be applied during the mapping process. Currently, there are three valid options:

- **Load balancing for DFG**, described in section 7.2.2
- **ILP based load balancing**, described in section 7.2.3
- **Energy efficient mapping**, described in section 7.2.4

Configuring Mathematical Solver

Hint: The settings described in this section only affect ILP based algorithms!
The section ***Solver Settings*** allows to configure the solver which is used to approximate the ILP problems, specify the minimal accuracy of the found solution and activate the MPS file output of the - ready to solve - ILP problem.

- **Generate MPS file** will activate MPS file generation. The resulting MPS file will contain the actual ILP problem for the chosen mapping strategy.

- **Max. Gap** specifies the maximal gap (percentage) between the LP relaxion and a feasible solution before the solver considers it to be optimal.
  Setting this value to 0.0 will order to solver to continue until either the final solution reaches the same value as the LP relaxion or another limit (below) has been reached while 1.0 will consider the first feasible solution being optimal.
  Valid values: $0.0 - 1.0$

Furthermore, it is possible to specify the maximum number of iterations or time spend on finding an optimal solution.

- **Max. Iterations (Abort)** specifies the maximal ***total*** number of iterations which may be performed by the approximation algorithm in order to find an optimal solution for the ILP problem.
- **Max. Time (Abort)** specifies the maximal ***total*** amount of time (milliseconds) that may be spend by the approximation algorithm in order to find an optimal solution for the ILP problem.
- **Max. Iteration (Suffice)** specifies the maximal number of iterations which may be performed by the approximation algorithm in order to ***improve*** a previously found feasible solution.
- **Max. Time (Suffice)** specifies the maximal amount of time (milliseconds) that may be spend by the approximation algorithm in order to ***improve*** a previously found feasible solution.

Setting one of these values to zero will pass the value of INT_MAX to the solver, technically removing the respective constraint.

7.3.2 Generating a mapping

Depending on the selected mapping strategy, it may be required to create tasks in advance of the mapping algorithm. The method **'Create Tasks'**, which is accessible through the *AMALTHEA Software Models* file context menu (right click on $*.amxmi$ and $*.amxmi\text{-}sw$ files), is capable of transforming partitioned[3] software models into software models with tasks.

The mapping can be performed once input models with the required amount of information are present[4]. Opening the context menu again (right click on $*.amxmi$ and $*.amxmi\text{-}sw$ files) and selecting **'Perform Mapping'** will open the 'Perform Mapping GUI' (see Figure 7.3).
The fields within the GUI are described below.

[3] A detailed guide on partitioning a model is available in Chapter 6
[4] see section 7.2 for detailed information about the mapping strategies and their required amount of information

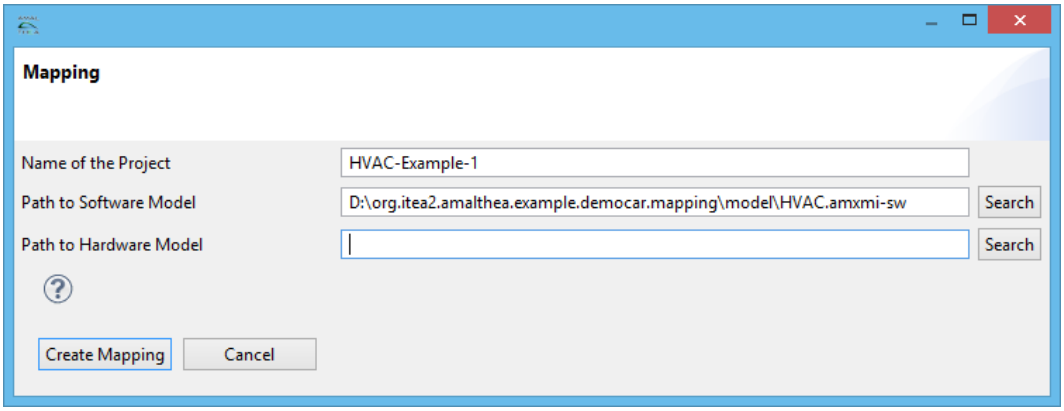

Figure 7.3: Perform Mapping GUI

- **Name of the Project** is automatically filled and based on the selected software model. This value is used to specify the base name of the output files and may be customized.
- **Path to Software Model** points to the location of the file containing the AMALTHEA Software Model which will be used during the mapping process. This path is automatically set, but may be changed if desired.
- **Path to Hardware Model** points to the location of the file containing the AMALTHEA Hardware Model which will be used during the mapping process.

Source Model	Element	Description
HW	`Core`	A `Core` represents the target of an allocation, an OS Model with a `Scheduler` for each `Core` will be generated.

	`CoreType`
	`Prescaler`
	`Quartz`
		A `Core`s `Prescaler`, the referenced `Quartz` and the `CoreType`s attribute `CyclesPerTick` of a `Core` are used to determine the number of processed `Instructions` per second.
		The `CoreType`s attributes (`DoubleValue`) starting with the label *EnEf-Volt_{SomeID}* and *EnEf-Scale_{SomeID}* are used to specify the voltage levels, i.e. the performance of a core during a specific voltage.
		Example: The attributes (values are in brackets) EnEf-Volt_HIGH (1.15), EnEf-Scale_HIGH (1.0), EnEf-Volt_LOW (1.05) and EnEf-Scale_LOW (0.5) specify the two voltage levels
		high with 1.15 Volt and full performance (Scale 1.0) and
		low with 1.05 Volt and half performance (Scale 0.5)

SW	`Runnable`	`Runnable`s will be distributed on the `Core`s (over the `Core`s `Scheduler`), their attribute `Instructions` is used during the load calculation for each `Core`
	`Activation` (`Periodic`)	The `Periodic` activation specifies the recurrence of the `Runnable`. The lowest recurrence is used to specify the overall deadline of all `Runnable`s, i.e. the max amount of time for the sum of all `Runnable` executions.
Constraints	`Process-Runnable-Group` `Runnable-Sequencing-Constraint`	Are used to determine the executional order of the `Runnable`s as well as their interdependencies

Mapping	`Runnable-Allocation`
	`Custom-Property`
	`LongValue`
		Specifies the selected voltage level and the number of `ExecutionCycles` at this voltage level. The annotation is defined as followed:
		Key: EnEf-VoltageLevel_{SomeID} (String)
		Val: No. of the `Runnable`s cycles at this voltage level (long)

Table 7.3: Processed information by mapping strategy for minimizing energy consumption

8 Building

This chapter describes how the outputs of the previous steps are processed in order to build an executable, which runs on a multi-core target platform. Two main steps are addressed in AMALTHEA. First, generating code from the architectural models, and second, generating an OSEK configuration.

8.1 Concept

The previous chapters have shown the modeling of software architecture and behavior (Chapter 5), the subsequent partitioning of runnables into tasks (Chapter 6), and the mapping of tasks onto cores (Chapter 7).

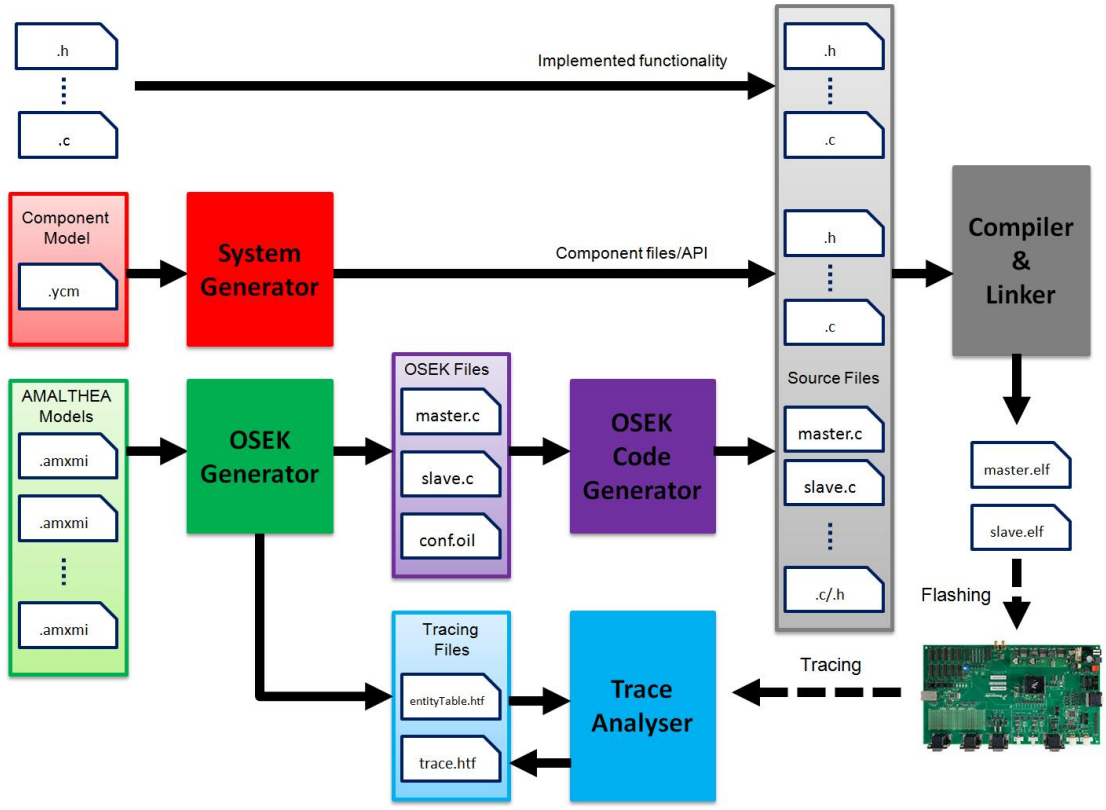

Figure 8.1: Overview building and integration process

Two remaining main jobs have to be done before the final creation, i.e., compiling and linking, of the binary files can be started:

- the generation of code and header files implementing the information contained in the component model,
- the configuration and generation of an operating system for managing and scheduling the execution of the tasks on the target.

The overall building process is depicted in Figure 8.1. A system generator (described in Section 8.2.1) generates code and header files for the component behavior, the system data structure and APIs for the access to this structure. An OSEK generator or, to be more precise, an OSEK configuration generator (described in Section 8.2.2) creates the main files for both cores and the configuration file for the OSEK. Out of these, in a preprocessing step typical for all OSEKs, additional source files are generated. The source files, including those who implement the functionality, are then compiled and linked. In the case of the HVAC demonstrator the OSEK is the *Erika Enterprise* operating system with the Eclipse-based tool *RT-Druid* [19]. RT-Druid executes the preprocessing step and the final compilation.

The resulting ELF files can be flashed onto the target processor. Also shown in Figure 8.1 is the possibility to trace software events on the running hardware. This is further elaborated in the following chapter.

The interactions between the artifacts generated in the processes described above are shown in Figure 8.1.

The operating system manages the tasks by scheduling them. From within each task runnables are called which implement the internal behavior of one or more components. This behavior consists mainly of calling the APIs for read and write accesses to the system data structure (resp. the port accesses of the component) an calls of functions who work on this data, i.e. the actual functionality.

8.2 Implementation

8.2.1 System Generator

In Chapter 5 it has already been explained, how the architecture of an embedded multi-core system is modeled with the YAKINDU CoMo component model. This system model is used by the YAKINDU System Generator to generate the source code for a CoMo system with the C programming language. The generator is a commercial development of the AMALTHEA project partner itemis and not part of the published tool chain.

A feature of the generated system code is the independence of the operating system on which it should run. It serves as a system API which can be used by operating system specific elements.

The generator creates a header-file for each type of component used in the system. Within the header-file for each component-port corresponding interface-functions are defined, which can be used in the implementation of the component to retrieve data from

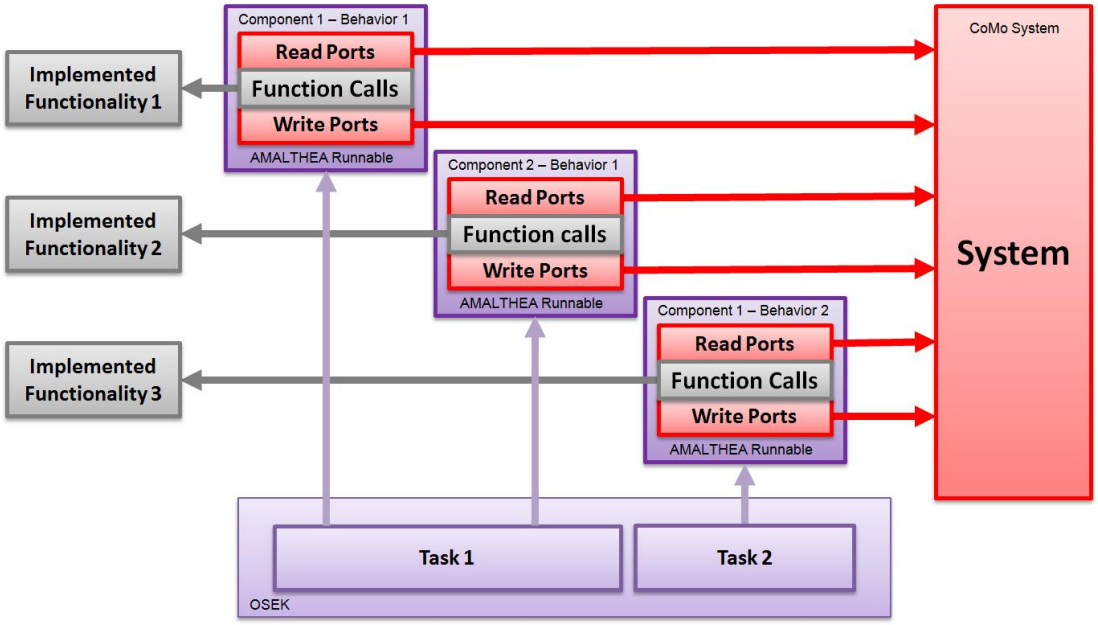

Figure 8.2: Example function and API calls between component behaviors, system, and OSEK

the system or to write data into the system. The amount and type of these interface-functions depends on the port type ('providing' or 'requiring') and on the elements within the referenced Franca IDL interfaces. In addition for each behavior of a component a function is created by the generator in which the actual functionality or behavior must be implemented. This implementation can exchange data with the system via the aforementioned interface-functions of the ports.

For a YAKINDU CoMo system, the generator creates a header- and a source-file. Within these files the basic system data structures are defined and the previously mentioned interface-functions of the components ports are implemented, which provides the accessibility to the system data structures for the components.

In addition, for each behavior of a component a corresponding function is generated. For example these functions can be called from an operating system task. Furthermore for Franca broadcasts appropriate callback functions are generated, which can be used to map the Franca IDL broadcasts to operating system events. This approach makes it possible to trigger the execution of a behavior by an operating system event.

In case of the HVAC demonstrator the operating system code, generated by the OSEK generator, uses the CoMo system code by executing the behavior functions of the components within respective OSEK tasks.

8.2.2 OSEK Generator

The OSEK Generator produces an OIL file from the software model. OIL stands for OSEK Implementation Language and is used to describe an OSEK operating system configuration. The OIL language employs objects to configure the different elements of the operating system. Table 8.1 shows these objects and their transformation from the AMALTHEA models.

OSEK	**AMALTHEA**
CPU	Base container for all OSEK definitions. A special MASTER_CPU definition is used to define the master core. Additionally for each core from the target hardware, a CPU_DATA object has to be set.
OS	Definitions of the operating system. For the Erika Enterprise OS, several EE_OPT objects are used to define the proprietary elements of the operating system.
APPMODE	Defines different running modes for the operating system. There is no special configuration from the AMALTHEA models. This object is user-defined.
ISR	ISRs (Interrupt Service Routines) from the mapping model, referenced in the software model. ISRs are derived from `Process`.
RESOURCE	A resource can be used from a task. Resources can be mapped from semaphores.
TASK	Tasks from the mapping model are referenced from the software model.
COUNTER	A counter is a hardware or software tick-source for alarms. Counter are represented through a stimulus form the stimulus model.
EVENT	The event model supports several `Events`, for example `ProcessEvent` to activate a process. `Events` are in the software model defined as OS Events.
ALARM	Alarms base on counters and can activate a task set, an alarm, or call an alarm callback routine. Alarms are set in the `CallSequence` of a task in the software model.

Table 8.1: OIL-AMALTHEA Mapping

The configuration is generated for the Erika Enterprise OSEK operating system. Input to the OSEK Generator is a template which contains non-standardized, general definitions specific to this operating system, which has to be written by hand. The OSEK Generator adds all the specific information, which is related to the software system that is build. In particular, the tasks generated in the partitioning process are transformed into OSEK tasks.

The target system has two cores. Each core has to be defined for the Erika Enterprise operating system. The following listing 8.1 shows the core definitions in the OIL file.

```
1  CPU_DATA = PPCE200ZX {
2    ID = "master";
3    MODEL = E200Z6;
4    APP_SRC = "master.c";
5    ...
6  };
7
8  CPU_DATA = PPCE200ZX {
9    MODEL = E200Z0;
10   ID = "slave";
11   APP_SRC = "slave.c";
12   ...
13 };
```

Listing 8.1: Core Definition in OIL

Additionally, there is a mapping for the used source files. These files are generated by the OSEK generator and contain the tasks with the respective runnables generated from the `CallSequence`.

The tasks are mapped to a scheduler which is referenced in the hardware model. Figure 8.3 illustrates a mapping from a task to a scheduler. Then, a scheduler in the hardware model is associated to a core from the target platform.

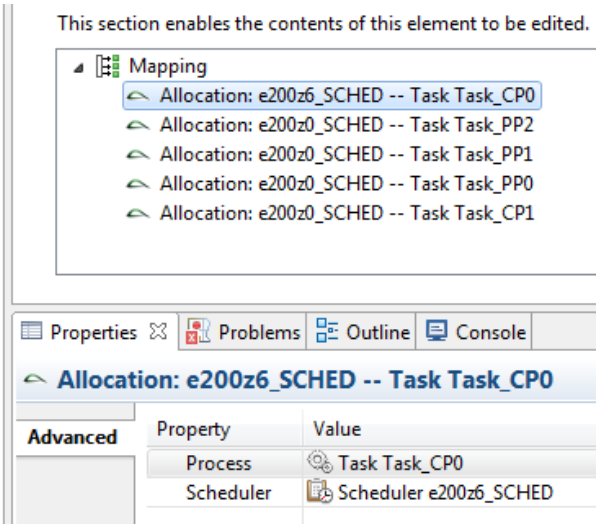

Figure 8.3: Mapping of Task to Scheduler

Listing 8.2 shows a generated OSEK task in the OIL file. The task contains an assignment to the core on which the task should run (CPU_ID).

```
1  TASK Task_CP0 {
2    CPU_ID = "master";
3    PRIORITY = 0;
4    AUTOSTART = FALSE;
```

```
5     STACK = SHARED;
6     ACTIVATION = 0;
7     SCHEDULE = NON;
8   };
```

Listing 8.2: Generated Task in OIL

9 Tracing

In the following chapter the tracing will be explained. Tracing is the concept of recording the software behavior of an embedded system during runtime.

9.1 Concept

In AMALTHEA, the tracing procedure typically follows four major steps:

- Recording trace data at runtime
- Save traces in Hardware Trace Format (HTF)
- Trace conversion from HTF to OT1
- Analyze with TA-Toolsuite

For the representation of the recorded tracing information, the Hardware Trace Format (HTF) is used. More detailed information on the HTF is given below.

9.1.1 Hardware Trace Format

The HTF (Hardware Trace Format) is a result of the AMALTHEA project. It is a compact trace format for the tracing of events on embedded systems, including multi-core systems. Its binary representation shortens the execution time of the tracing instructions and improves the storage efficiency of the tracing data on the target system, compared to other trace formats. On the host system, the target traces can be saved in a **.htf* file without the need for any extra conversion. Still, it is compatible to other trace formats (e.g. BTF[1], OT1[2]), so a conversion into these formats is possible.

HTF files are divided into three sections. The first part is the header where meta data can be found. Exemplary parameters are the creation date, a description and the target system name. In the second section, reference tables are defined. These tables represent the association between the recorded events and the AMALTHEA software components. Every table entry consists of a specific hex value and a textual description. Subsequently, reference tables allow the interpretation of the binary trace data. The last section of the HTF file contains the recorded trace datasets. Each HTF trace dataset consists of a timestamp, an entity, and an event. When tracing single core systems, this part has one data section, whereas the trace of multi-core systems can produce additional data sections.

[1] http://wiki.eclipse.org/Auto_IWG#Publications
[2] https://gliwa.com/ot1

Header

The header provides the global information of a tracing session. Each entry is headed by a hash symbol for a simplified parsing of the file. The following list contains all defined parameters of the header, each with an explanation and a brief illustrative example.

Format: The first parameter is the *Format*, which has to have the value "HTF" in a valid HTF file.
Example: `#Format HTF`

Version: The parameter *Version* shows the version number of the used HTF specification.
Example: `#Version 1.0`

URL: The parameter *URL* references a website where the HTF specification document can be found.
Example: `#URL http://wiki.eclipse.org/Auto_IWG#Publications`

Project: The *Project* parameter provides information about the project context of the tracing session. For instance, the project, the respective department, or the institution can be mentioned.
Example: `#Project ITEA2 AMALTHEA`

TargetSystem: Information about the target system, on which the tracing was done, is saved in the parameter *TargetSystem*.
Example: `#TargetSystem Freescale MPC5668G evaluation board`

Description: In the parameter *Description* a short textual description of the software can be given. Conceivably, the application name with some special features of the program can be put in.
Example: `#Description Test-Application with 7 tasks and 2 interrupts`

NumberOfCores: The *NumberOfCores* parameter saves how many of cores of the target system are traced.
Example: `#NumberOfCores 2`

CreationDate: The parameter *CreationDate* refers to the point in time when the tracing session was started. The value contains the date as well as the time and requires the following format:
Year-Month-Day Hour:Minute:Second (yyyy-mm-dd hh:mm:ss)
Example: `#CreationDate 2014-04-04 13:15:25`

TimeScale: The *TimeScale* parameter specifies the unit of the timestamps in the recorded trace. Applicable time units in an HTF are:

- picosecond (ps)
- nanosecond (ns)
- microsecond (us)
- millisecond (ms)
- second (s)

Example: `#TimeScale ns`

TimeScaleNumerator and TimeScaleDenominator: In order to generate timestamps scaled in the base unit of time (ps, ns, us, ms, s), the timestamps of the CPU must be converted. The *TimeScaleNumerator* and *TimeScaleDenominator* parameters are provided for this purpose. Via the *TimeScaleNumerator* a multiplier is defined, while the *TimeScaleDenominator* provides a divisor. The CPU timestamps are multiplied with the *TimeScaleNumerator* and then divided by the *TimeScaleDenominator*. The result is a modified timestamp with the required resolution.
Example:

```
#TimeScaleNumerator 4
#TimeScaleDenominator 1
```

TimestampLength: The parameter *TimestampLength* defines the byte size of the timestamps. The value represents the number of used bytes. The following table provides the maximum values for systems with 8-bit, 16-bit, 32-bit, and 64-bit architectures.

Number of bytes	Maximum in hexadecimal	Maximum in decimal
1	FF	256
2	FF FF	65.536
4	FF FF FF FF	4.294.967.296
8	FF FF FF FF FF FF FF FF	18.446.744.073.709.600.000

Table 9.1: HTF maximum values of timestamps

Example: `#TimestampLength 4`

EntityLength: The parameter *EntityLength* shows the number of bytes which an entity ID consists of.
Example: `#EntityLength 2`

EventLength: The parameter *EventLength* shows the number of bytes which an event ID consists of.
Example: `#EventLength 1`

Reference tables

In the reference tables a hexadecimal value gets linked with a string. Hereby, the recorded trace can be transformed into a legible format. Again, the hash sign is used as a starting symbol, while a blank character separates the hexadecimal values from the referenced string.

TypeTable: System events can be generated or invoked by diverse entities. For example, operating system behaviors, the events of tasks, and interrupt service routines can be recorded. The *TypeTable* assigns IDs to the entity types. An exemplary *TypeTable* is shown in the extract of an HTF file below.

```
#TypeTable
#-00 Task
#-01 ISR
#-02 Runnable
#-03 CodeBlock
#-04 Signal
#-05 Semaphore
```

EventTables: In the following reference tables the events are assigned to the respective entity types.

```
#TaskEventTable
#-00 activate
#-01 start
#-02 resume
#-03 preempt
#-04 terminate
#-05 wait
#-06 release
#-07 poll
#-08 run_polling
#-09 park
#-0A poll_parking
#-0B release_parking

#ISREventTable
#-00 start
#-01 resume
#-02 preempt
#-03 terminate

#RunnableEventTable
#-00 start
#-01 suspend
#-02 resume
#-03 terminate

#CodeBlockEventTable
#-00 start
#-01 stop

#SignalEventTable
#-00 read
#-01 write

#SemaphoreEventTable
#-00 lock
#-01 unlock
```

EntityTable and EntityTypeTable: In the *EntityTable* each software entity gets an ID, while in the *EntityTypeTable* a type is assigned to each entity. The following extract of an HTF file shows an example for both reference tables.

```
#EntityTable                    #EntityTypeTable
#-0000 TaskZ6One                #-0000 00
#-0001 TaskZ6Two                #-0001 00
#-0080 TaskZ0One                #-0080 00
#-0081 TaskZ0Two                #-0081 00
#-0082 TaskZ0Three              #-0082 00
#-0083 TaskZ0Four               #-0083 00
#-00EE Z6_interrupt             #-00EE 01
#-00FF Z0_interrupt             #-00FF 01
```

Tracedata

In the last section of the HTF file, the binary tracing data are filed. The beginning of this section is marked with the keyword *TraceData*. After that keyword, the hash symbol is only used to mark the beginning of a tracing session for each core. Every line contains one dataset and an optional comment. Each dataset is a hexadecimal number. This number comprises three columns which contain the information concerning the timestamp, the entity ID, and the event ID. The actual width of each column is defined by the values of *TimestampLength*, *EntityLength*, and *EventLength*. Every comment starts with a "//". A example trace of a dual-core system is shown below.

```
#TraceData    //| 4 Byte Timestamp  | 2 Byte Entity ID | 1 Byte Event ID |
#-01         // Core 1
00000000000000 //Timestamp: 00000000  Entity: 0000->T1  Event: 00->activate
00000064000001 //Timestamp: 00000064  Entity: 0000->T1  Event: 01->start
00002710000100 //Timestamp: 00002710  Entity: 0001->T2  Event: 00->activate
00002774000003 //Timestamp: 00002774  Entity: 0000->T1  Event: 03->preempt
00002780000101 //Timestamp: 00002780  Entity: 0001->T2  Event: 01->start
0000417E000104 //Timestamp: 0000417E  Entity: 0001->T2  Event: 04->terminate
000041E2000002 //Timestamp: 000041E2  Entity: 0000->T1  Event: 02->resume
00004EE7000004 //Timestamp: 00004EE7  Entity: 0000->T1  Event: 04->terminate

#-02         //Core 2
00000000000200 //Timestamp: 00000000  Entity: 0002->T3  Event: 00->activate
00000055000202 //Timestamp: 00000055  Entity: 0002->T3  Event: 02->start
00450145000205 //Timestamp: 00450145  Entity: 0002->T3  Event: 05->terminate
```

9.2 Implementation

9.2.1 Trace Generation on Platform

The trace generation on the target system is organized in a way, that besides the application tasks three sections are implemented for the tracing functionality.

- Trace Generator
- Trace Manager
- Trace Transmitter

These three sections are briefly explained below.

Trace Generator

The blocks in the source code where trace datasets are generated are part of the trace generator. These blocks are called *tracepoints* and are put into effect with the command *putTrace()* in this implementation. Figure 9.1 illustrates the methods containing these *tracepoints*. On the upper left, Figure 9.1 shows how the events are traced by the schedule

Figure 9.1: HTF trace generator in the embedded trace concept

using the *preTaskHook* and *PostTaskHook* functions of the operating system. On the upper right is shown how the Interrupt Service Routines are traced, while at the beginning and end of each routine a *tracepoint* is set. The task events *activate* and *terminate* are entered via additional functions, which is shown in the bottom part of the figure.

Trace Manager

The trace manager is accountable for the management of the tracing datasets. This task can be divided into three sub-tasks which are collecting, caching and providing. Figure 9.2 shows a typical workflow of the trace manager. Figure 9.2 describes how the collector

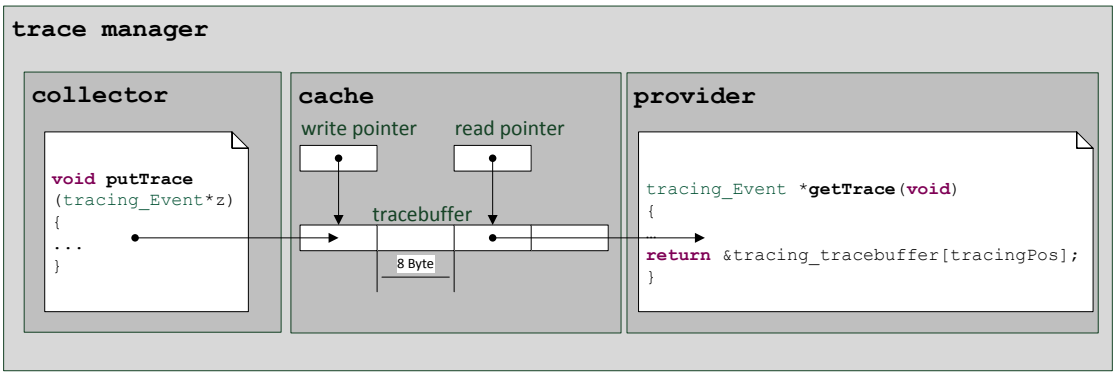

Figure 9.2: HTF trace manager in the embedded trace concept

lists each event in a *tracebuffer*. Moreover, it is shown how the provider reads the data from the cache by the use of a *read pointer*.

Trace Transmitter

The trace transmitter covers the last field of tasks of the tracing functionalities. The transmitter uses the provider function *getTrace()* in order to read traces from the cache. The loaded data sets are provided sequentially by using the serial interface. The following figure illustrates the outlined process (see Figure: 9.3).

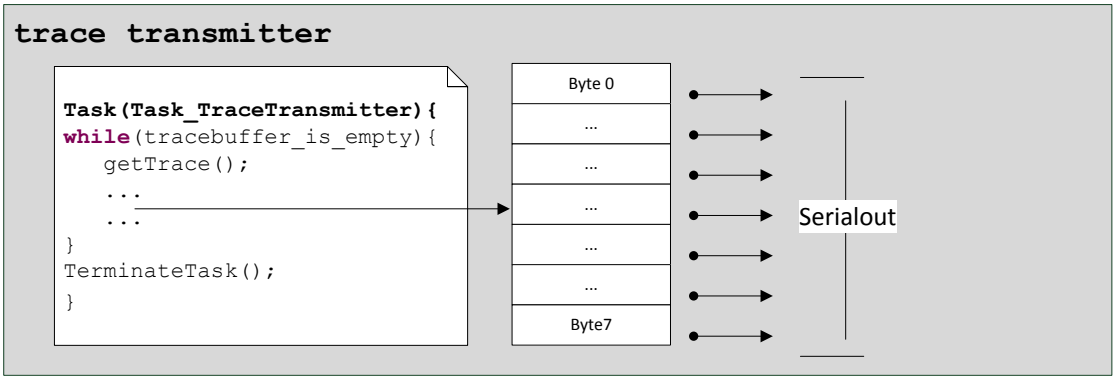

Figure 9.3: HTF trace transmitter in the embedded trace concept

9.2.2 Trace Receiver

The trace receiver is a plugin of the AMALTHEA Tool Platform. The configuration of the recording can be set up in a dialog box. The following screenshot shows the user interface of the trace receiver (see Figure 9.4). In this case, the software behavior of the HVAC demonstrator was traced.

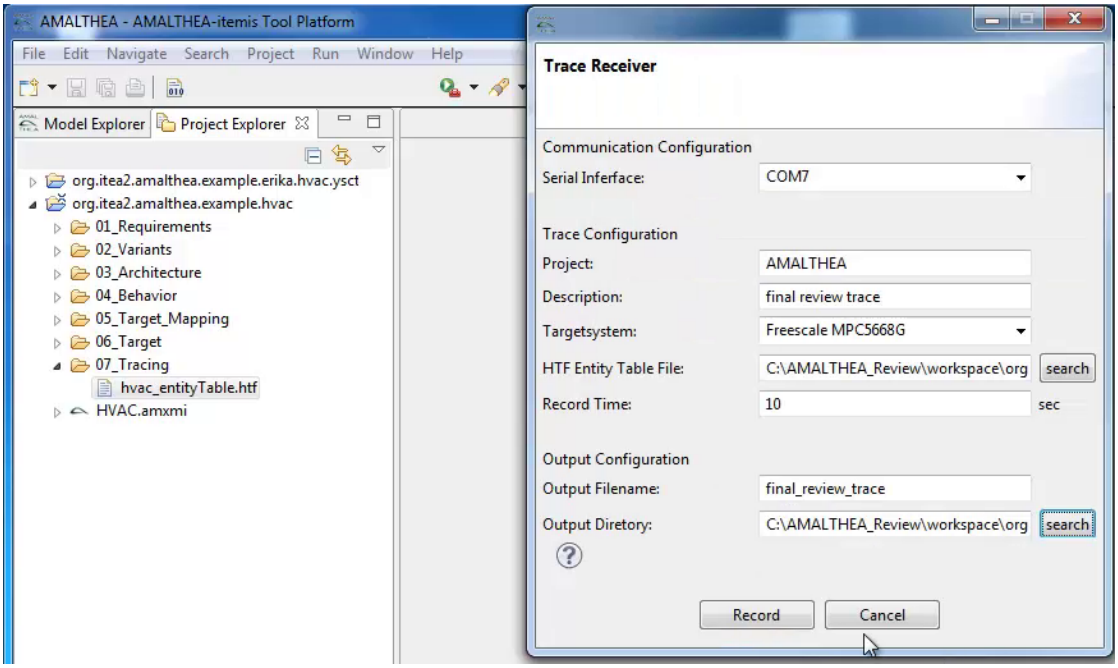

Figure 9.4: Screenshot of the AMALTHEA trace receiver plugin

9.2.3 Trace Conversion

Conversion of trace data from HTF to OT1 is implemented as Eclipse plug-in by means of two projects:

traceconverter.ot1 contains the OT1 meta-model described by means of EMF. It is based on the OT1 model description provided by Gliwa GmbH, e.g. as XML schema, at [13].

traceconverter.converter implements the major functionality of the trace converter. The implementation is split into three files separating handler functionality of the Eclipse plug-in, mapping of HTF-to-OT1 model elements, and conversion of trace data read from HTF file to OT1 format.

In contrast to the OT1 meta-model description that is provided in a separate project, HTF elements and keywords are defined in the class *Htf2Ot1Converter* implemented in the file *Htf2Ot1Converter.java*. Keywords allowed or even required within the HTF

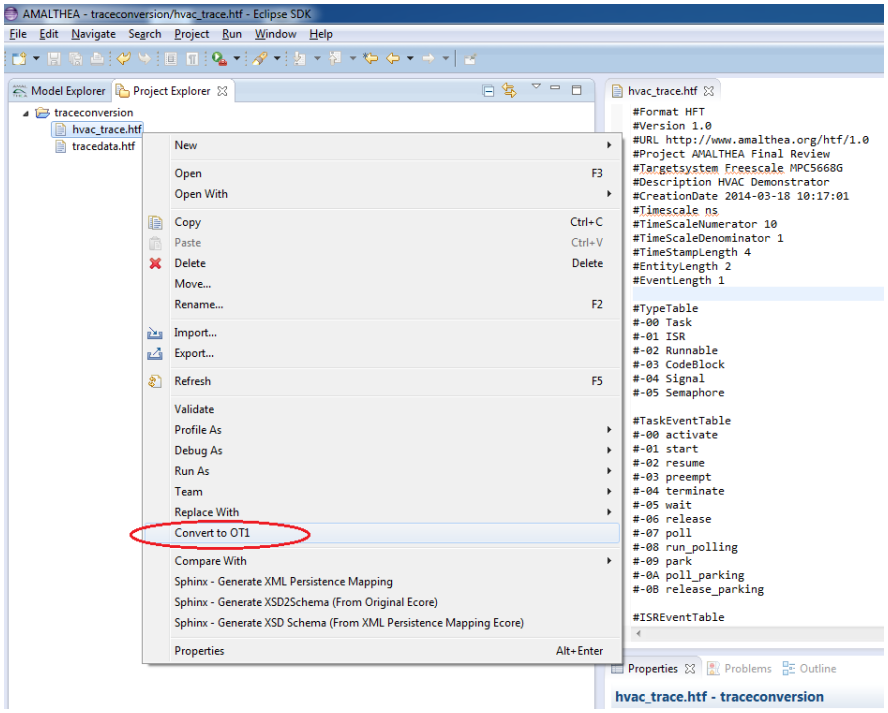

Figure 9.5: Menu item to convert trace data from HTF to OT1 data format.

header as well as entity types, type-specific events, and further keywords defined within HTF specification are defined by means of *enum*. This way, a new revision of the HTF specification can easily be implemented by adapting the corresponding *enum* definitions within the class *Htf2Ot1Converter*. But this only covers modifications of the single model specifications. To enable an easy adaption of the convert functionality — provided by the trace converter plug-in — in case of modifications of the HTF as well as OT1 format, the mapping of HTF model elements to OT1 model elements is encapsulated into a separate file named "*Htf2Ot1Mapping.java*". Here static Map-Objects are defined, one covering mappings of HTF events to OT1 events and another covering mapping of HTF entity types to OT1 entity types.

Since this plug-in converts trace data provided by HTF files to the OT1 file format, the context menu item "*Convert to OT1*" is only available if a HTF file is selected within the Project Explorer (cf. Figure 9.5).

The output of the implemented trace converter plug-in is a XML-file according to the XML schema of the OT1 format provided by Gliwa GmbH [13]. The output file is named as the HTF input file and differs only with respect to its file extension. Thus, converting trace data from a HTF file "*test.htf*" results in an output file named "*test.xml*".

After the conversion of the trace into the OT1 format, the *TA-Toolsuite* can open the trace file. This toolsuite allows visualizing the traces with a Gantt chart view. Along with the graphic presentation, the TA-Toolsuite offers the possibility of computing opti-

mization suggestions for the software. Figure 9.6 shows the Gantt chart of the recorded software of the HVAC demonstrator.

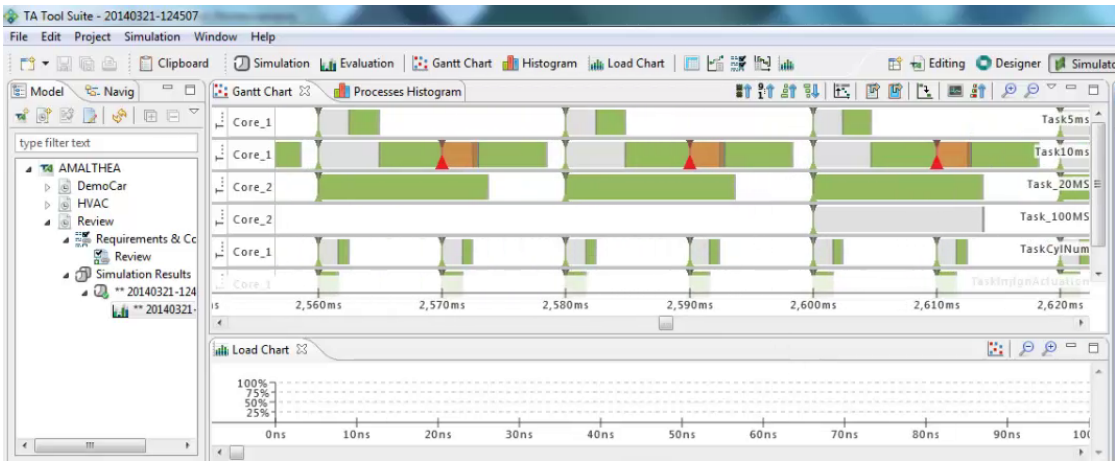

Figure 9.6: Screenshot of the TA-Toolsuite with the Gantt chart view

10 Conclusion

Within this deliverable "D3.4 Prototypical Implementation of Selected Concepts", we described concepts that have been developed and prototypically implemented in the context of the AMALTHEA project. To prove the applicability of these concepts, the Heating, Ventilation, and Air Conditioning (HVAC) case study was carried out. Based on the AMALTHEA Tool Platform, we built a customized toolchain within the HVAC case study using external tools as well as tools developed within the AMALTHEA project. All these tools were integrated into the AMALTHEA Tool Platform to enable a continuous software development for an embedded multi-core system.

Using the HVAC as an example, we presented all steps covered by the AMALTHEA Tool Platform described in Section 2. In detail, we first presented requirement engineering and variability modeling for our HVAC case study. Afterwards, we described the software architectural and behavioral modeling, i.e. specification of interfaces, components, and their behavior. The following steps, namely partitioning, mapping, code generation, and tracing, were realized by AMALTHEA tools.

The methodological aspects of architectural modeling, partitioning, mapping (including hardware modeling), and tracing, the implementation in terms of Eclipse plugins, and the HVAC case study on an industry scale multi-core system are the main contributions of Work Package 3 to the AMALTHEA project.

Bibliography

[1] AMALTHEA: *Deliverable 3.2: Hardware model specification and examples of hardware models.* January 2013

[2] AMALTHEA: *Deliverable 4.2: Detailed design of the AMALTHEA data models.* January 2013

[3] AMALTHEA: *Deliverable 4.4: Report on Model and Tool Exchange.* April 2014

[4] Amdahl, Gene M.: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* 1967

[5] Bellman, Richard: On a Routing Problem. In: *Quarterly of Applied Mathematics* 16 (1958), S. 87–90

[6] Brink, Christopher ; Kamsties, Erik ; Peters, Martin ; Sachweh, Sabine: On Hardware Variability and the Relation to Software Variability. In: *40th Euromicro Conference on Software Engineering and Advanced Applications.* Verona, Italy, 2014

[7] consortium, Amalthea: *www.amalthea-project.org.* Webpage. April 2014

[8] Cormen, Thomas H. ; Stein, Clifford ; Rivest, Ronald L. ; Leiserson, Charles E.: *Introduction to Algorithms.* 2nd. McGraw-Hill Higher Education, 2001. – ISBN 0070131511

[9] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. In: *NUMERISCHE MATHEMATIK* 1 (1959), Nr. 1, S. 269–271

[10] Drozdowski, Maciej: *Scheduling for Parallel Processing.* Springer, 2009 (Computer Communications and Networks). – i–xiii, 1–386 S. – ISBN 978-1-84882-309-9

[11] 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

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

[13] Gliwa: *Open Timing 1.* November 2012. – `http://www.gliwa.com/ot1/index.html`

[14] Gomaa, H.: A Software Design Method for Real-time Systems. In: *Commun. ACM* 27 (1984), September, Nr. 9, S. 938–949. – ISSN 0001-0782

[15] Grigoriev, Alexander ; Sviridenko, Maxim ; Uetz, Marc: Machine scheduling with resource dependent processing times. In: *Mathematical programming* 110 (2007), Nr. 1, S. 209–228

[16] Krawczyk, Lukas ; Kamsties, Erik: Hardware Models for Automated Partitioning and Mapping in Multi-Core Systems using Mathematical Algorithms. In: *International Journal of Computing* (2014)

[17] Kwok, Yu-Kwong ; Ahmad, Ishfaq: Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. In: *IEEE Trans. Parallel Distrib. Syst.* 7 (1996), Mai, Nr. 5, S. 506–521. – ISSN 1045-9219

[18] Sohi, Gurindar S. ; Vijaykumar, T.N. ; Keckler, Stephen W. (Hrsg.) ; Olukotun, Kunle (Hrsg.) ; Hofstee, H. P. (Hrsg.): *Multicore Processors and Systems.* 2009

[19] Srl, Evidence: *Erika Enterprise and RT Druid.* April 2014. – URL `http://erika.tuxfamily.org/drupal/`