PADERBORN UNIVERSITY
The University for the Information Society

Faculty of Computer Science, Electrical Engineering and Mathematics

Early Performance Analysis of Automation Systems Based on Systems Engineering Models

Jens Frieben

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Paderborn, August 26, 2017

Abstract

Designing and scaling complex, networked automation systems is a challenging task. Developers have to consider different kinds of influence factors that impact the system performance already in the early stages of the development. For example, estimating the average workload of a Programmable Logic Controller (PLC), deploying software onto resources, or assigning sensors and actuators to PLCs are difficult tasks when coping with complex systems. If performance bottlenecks are detected too late, costly corrections may follow and the commissioning can be significantly delayed.

There already exist several approaches to predict the performance of a system under development. However, the developers must identify and decide themselves which factors must be regarded and to what detail. Therefore, developers usually require in-depth knowledge of the system which is – at least to this level of detail – not available until the discipline specific development starts. Existing approaches only cover parts of the system and differ in their level of detail, scope, usability, and applicability to predict automation systems. In particular it is not clear, which factors influence the performance of a PLC and how these can be used for an early validation of a system model.

The first contribution of this thesis is the identification and creation of a list of influence factors that impact one or more quality of service attributes of an automation system. For each factor, a decision is made, whether it is available in the early development stages, what assumptions must be made by the developer, and what impact it has on the overall system performance.

The second contribution is a method that enables developers of automated systems to define the identified influence factors in systems engineering models and to carry out a subsequent automatic performance analysis. The method includes a model for the specification of influence factors and their parameters within an existing Systems Engineering model as well as a process to guide the developers through the specification of these factors. Based on these extended Systems Engineering models, an analysis can be performed to predict the utilization of a selected PLC. The results can be used to evaluate the design of the automation in the early phases and, therefore, avoid costly and time consuming changes in the following integration phase.

Zusammenfassung

Der Entwurf und die Auslegung komplexer, vernetzter Automatisierungssysteme ist eine anspruchsvolle Aufgabe. Während der Entwicklung dieser Systeme müssen unterschiedliche Arten von Einflussfaktoren berücksichtigt werden, die die Leistungsfähigkeit einer Speicherprogrammierbaren Steuerung (SPS) beeinflussen können. Eine präzise Schätzung der Auslastung einer SPS ist gerade in den frühen Phasen der Entwicklung eines automatisierten Systems nur schwer möglich. Diese wird zum Beispiel durch die Verteilung von Softwarekomponenten auf Ressourcen oder die Zuordnung von Sensoren und Aktoren zu Steuerungen beeinflusst. Werden jedoch Leistungsengpässe zu spät erkannt, können kostenaufwändige Korrekturen folgen und die Inbetriebnahme des Systems signifikant verzögern.

Zur Vorhersage der Performanz eines sich in der Entwicklung befindlichen Systems gibt es bereits mehrere Ansätze. Jedoch müssen die Entwickler selbst entscheiden, welche Faktoren für die Vorhersage entscheidend sind und zu welchem Detailgrad diese erfasst werden müssen. Daher benötigen die Entwickler in der Regel ein fundiertes Wissen über das System, dass - zumindest bis zu diesem Detaillierungsgrad - nicht bis zur disziplinspezifischen Entwicklung verfügbar ist. Zusätzlich decken bestehende Prognoseansätze oft nur Teile des Gesamtsystems ab und unterscheiden sich in ihrem Detaillierungsgrad, Umfang, ihrer Verwendbarkeit und Anwendbarkeit auf Automatisierungssysteme. Insbesondere ist unklar, welche Faktoren die Leistungsfähigkeit einer SPS beeinflussen und wie diese für eine frühzeitige Validierung eines Systemmodells genutzt werden können.

Der erste Beitrag dieser Arbeit ist daher die Identifizierung von Einflussfaktoren, die eine oder mehrere Qualitätseigenschaften eines Automatisierungssystems beeinflussen. Für jeden Faktor wird entschieden, ob dieser bereits in den frühen Entwicklungsstadien identifiziert werden kann, welche Annahmen vom Entwickler getroffen werden müssen und welche Auswirkungen er auf das Gesamtsystem hat.

Der zweite Beitrag dieser Arbeit ist eine Methode, die es Entwicklern automatisierter Systeme ermöglicht, die zuvor identifizierten Einflussfaktoren in Systems Engineering Modellen zu integrieren und automatische Leistungsanalysen durchzuführen. Hierzu gehören ein Modell zur Spezifikation der Einflussfaktoren und Parameter innerhalb eines bestehenden Systems Engineering Modells sowie ein Prozess, um die Entwickler durch die Spezifikation der Faktoren zu führen. Basierend auf den erweiterten Systems Engineering Modellen können automatische Analysen zur Leistungsprognose einer ausgewählten SPS durchgeführt werden. Mit den Ergebnissen der Prognose lassen sich schon in den frühen Phasen der Entwicklung Designentscheidungen validieren und somit spätere, kostspielige Änderungen vermeiden.

Danksagung

Danke. Dieses Wort möchte ich an alle aussprechen, die mich in den letzten Jahren mit Rat und Tat unterstützt haben. Anfangen möchte ich hiermit bei Wilhelm Schäfer. Ohne seinen Zuspruch und sein direktes Feedback wäre es wohl nicht zu dieser Arbeit gekommen. Seine Unterstützung, gerade in den ersten Jahren, hat mir sehr geholfen an dem Ziel der Promotion festzuhalten.

Danken möchte ich auch Eric Bodden und Roman Dumitrescu, die mich gemeinsam nach Wilhelm Schäfer weiter betreut und mir so geholfen haben diese Arbeit zum Abschluss zu bringen. Ich bedanke mich auch bei Ansgar Trächtler, der mir bei allgemeinen und automatisierungsspezifischen Fragen gute Antworten und Anregungen geben hat. Das Gleiche gilt für Matthias Meyer, der mir in vielen Diskussionen andere Blickwinkel aufgezeigt und mich über die letzten sechs Jahre immer unterstützt hat.

Einen besonderen Dank richte ich an Marie Platenius und Matthias Becker. Beide haben mir mit intensiven Diskussionen, Feedback und Tipps bei der Leistungsprognose, der Evaluierung und dem Aufbau sehr geholfen. Im allgemeinen danke ich allen Kollegen und Kolleginnen der Fachgruppe Softwaretechnik und des Fraunhofer IEM, welche mit mir in den letzten Jahren bei unzähligen Kaffeepausen Probleme und Ideen diskutiert haben. Insbesondere möchte ich mich bei Jörg Holtmann, David Schmelter, Thorsten Koch und meinem Bürokollegen Markus Fockel bedanken. Für technische oder administrative Fragen konnte ich mich immer an Jutta Haupt und Jürgen 'Sammy' Maniera wenden. Allen nochmals vielen Dank!

Einen guten Start in das Thema Leistungsprognose sowie Unterstützung bei der Realisierung verdanke ich Steffen Becker. Für die gute Zusammenarbeit mit dem Industriepartner Phoenix Contact danke ich insbesondere dem Projektleiter Henning Heutger.

Und natürlich möchte ich mich ganz herzlich bei meiner Familie bedanken! Zunächst bei meinen Eltern, die mir meine Ausbildung ermöglicht und damit die Grundlage für diese Arbeit geschaffen haben. In erster Linie möchte ich mich aber bei meiner Frau Tanja bedanken, welche mich die letzten Jahre gefördert, beraten, erduldet und auf vielfältige Weise unterstützt hat.

Contents

CHAPTER 1

Introduction

The size and the complexity of automated systems has grown rapidly in the last years and their use has extended to more domains than just the production of goods. Automation of tasks can be found in every aspect of day-to-day life, ranging from traffic control, wastewater processing, wind energy production, and the control of complex chemical processes as shown in Figure 1.1. These automation systems usually require a thorough planning in the early development stages due to the fact that it is usually not possible to create prototypes or make adjustments in the later phases of the development process [LFVH13, FT14].

(a) Traffic system (b) Wastewater (c) Wind energy (d) Chemical plant

Figure 1.1: Examples for automated systems (source: Phoenix Contact)

Automation tasks are usually performed by a combination of electrical and mechanical machines like conveyor belts, rollers, pushers, drills, or robots. They are controlled by Programmable Logic Controllers (PLC) which run software to compute the tasks at hand [FMS04, GG13, Lun08, KRK95]. Automation systems are, depending on size and purpose, hierarchically structured with coordination tasks on higher levels and critical, real-time tasks on lower levels. An example for such a hierarchy are several conveyor belts in an automation system which are controlled by a top level synchronization system. On the lower levels (belt control), motion control requires the PLC to read sensor data, perform the computation, and send commands back to the motors in less than 1 ms to ensure smooth operation. When planning an automation system, this mixture of top level decision making and low level real-time communication needs to be carefully planned. Controlling different automation tasks puts a high load on the PLC, leading to a high utilization of the CPU. Thus, for each automation system and environment, it must be ensured that **the PLC can cope with the load of data to be processed**, such that timely operation is guaranteed. However, selecting a fitting PLC for the automation task or adapting the overall structure of the system **is mostly based on the experience of the developers** and might lead to incorrect assumptions and estimations.

These developers are usually members of different disciplines involved in the development of automation systems [VHSFL14, KVH13, FEH$^+$13, BR05]. Process engineers define the working steps (drop, align, drill, realign, bend etc.) of the machine to produce the final product. Mechanical engineers construct the shape of the production system. Control engineers have to develop and parameterize control algorithms (e.g., proportional-integral-derivative (PID)), which calculate the data for actuators. The electrical engineers have to set up the PLC, sensors, actuators, and fieldbusses. A software engineer programs the sequence control and interfaces to overlying management systems. Each of these disciplines impact the overall design of the automation system and consequently the **factors that influence the utilization** of a PLC. In this thesis, the term **influence factor** is used to categorize factors that will directly or indirectly impact the utilization of a PLC.

The use of Systems Engineering (SE) methods is becoming a necessary prerequisite to develop large and complex automation systems [VHSFL14, TDBG15, BOF$^+$14]. Systems Engineering models provide a rough sketch of the system under development. They are used to improve the overall understanding of the system and to coordinate the involved disciplines by providing initial, coarse models of the system under development. These models are the basis for the subsequent, fine grained and discipline-specific development stages. By using these initial system models, changes in one discipline – for example switching the fieldbus type, splitting functionality onto multiple PLCs, or adding new sensors and actuators – can be better identified and tracked [GGS$^+$07, GCD$^+$14].

The design decisions made in the early stages of the development are often critical for the following, discipline specific stages [HSST13]. However, changes in one discipline will most likely influence or affect design decisions and artifacts of the other disciplines as well. A control engineer choosing a fitting PLC with a given set of features and properties, has an impact on the software development due to available services, programming languages, and libraries. The selection of the fieldbus influences the electrical design, may raise security issues, and impacts the overall performance of the PLC. Rearranging sensors and actuators might lead to a different software deployment or influences how many PLCs are needed to perform all necessary control tasks. Splitting software functions onto multiple PLCs may lead to more traffic on the fieldbus due to synchronization and data exchange, which in turn might reduce the minimum cycle time but increase the delay of messages.

Figure 1.2 shows a simplified development process based on VDI 2206 [Ver04]. During the system design phase, requirements are gathered and a coarse system model created. The different disciplines start their development based on this system model. During the integration phase, errors may occur due to unspecified requirements or lack of synchronization between the disciplines. The figure depicts the errors E1 and E2 and arrows indicate the phases in which these could have been avoided. To give an insight into the various problems that may arise during the development of an automation system, the errors E1 and E2 are briefly described in the following.

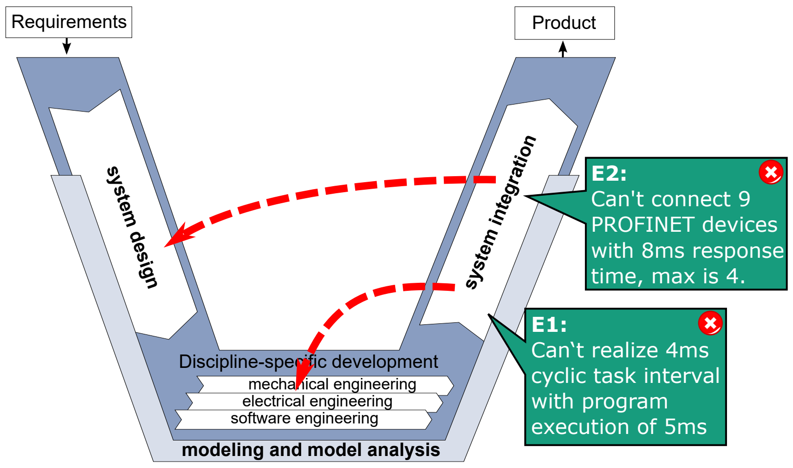

Requirements
Product
system design
system integration
Discipline-specific development
mechanical engineering
electrical engineering
software engineering
modeling and model analysis
E2: Can't connect 9 PROFINET devices with 8ms response time, max is 4.
E1: Can`t realize 4ms cyclic task interval with program execution of 5ms

Figure 1.2: Costly changes in the late phases of the development due to performance-related errors

E1: For motion control purposes, it is necessary to process data synchronously and in a very short amount of time. Therefore, cyclic tasks are often configured with a maximum interval time of 2 to 5 ms and setup for the execution of motion control programs. In case multiple programs are executed in the same task, other services require more computation time, or if some programs preempt the motion control program, the set interval time can not be met. As a result, the motion control will not work correctly and workpieces or the whole production plant might be damaged. A watchdog, checking the task interval time and the actual execution time of the programs will put the production plant into an error state if a certain limit has been reached. Therefore, it is essential to configure task times and program deployments correctly and foresee possible peaks in the PLC's utilization. If different developers are used to program the behavior of the production plant, they might not even know that these timing constraints exist.

E2: The second error describes a problem based on an incorrect dimensioning of bus systems and their throughput. A fieldbus is used to send sensor or actuator data between the PLC and the IOs. Depending on the fieldbus mode, the send- and receive interval is bound to the program/task execution. If a motion control task is running every 4 ms, reading sensor inputs from the devices and sending commands to the actuators, the data must also be sent every 4 ms from each device in the network. This will put stress on the PLC which needs to copy the messages from the IP stack to the program and back again. This copying of data might not sound like a very performance consuming task, but depending on the number of devices, the amount of data, and the necessary refresh interval of task and fieldbus, many PLCs reach their limit fast [FH12].

1.1 Problem Statement

The examples show that designing and scaling a complex, networked automation system is a challenging task. The developers have to consider different kinds of influence factors that have an impact on the overall system. Also, these factors are contributed by different disciplines as briefly described above. Flawlessly estimating the average workload of a PLC, deploying software onto resources, or assigning sensors and actuators to PLCs will become a difficult task when coping with such complex systems.

Miscalculations or other design errors identified in the late phases of the development can lead to costly changes or substantial delays for the commissioning of the automation system [SHK98]. Therefore, it will be mandatory to provide means for specifying automation-specific information that will help developers to evaluate their designs in the early phases of the development.

There already exist several approaches to predict the behavior and quality of service attributes of a system under development. These approaches usually cover one or more factors that influence the system but not every aspect of the automation domain. They are also only applicable during the discipline-specific development or the system integration phase. Figure 1.3 illustrates the four disciplines and maps selected tools to these areas (indicated by the numbers). These areas overlap to indicate that some approaches are designed to cover or require information from multiple disciplines. TrueTime [HCÅ03] (1) is a MATLAB [Mat16] and Simulink [Mat17] based simulator for real-time control systems. The framework allows the specification, programming, and simulation of programs, threads, real-time kernels, schedulers, network transmissions, and continuous plant dynamics. To check for the design of the network with regard to throughput, determinism, delays and jitter different network simulators like OMNeT++ [Ope17] (2), OPNET [Riv17] (3), or NS2/NS3 [VIN17, NS-17] (4) are available. Approaches tailored for automation systems are presented by Frey and Liu [LF07] (5), [MDFF06b] (6), or Halang [LF12] (7) to predict the response times of complex hierarchically structured systems with control loops based on different network types and topologies. Other approaches focus on general techniques to model and analyze complex systems for various domains like large scale server environments [BKR09, RBB+11] (8), automotive [TA17], or embedded devices [Obj06] (9), [Wan06] (10). Tools like SolidWorks [Das17] (11) are used to create plain mechanical CAD constructions of an automation system. With [ISG17] (12), simulations can be performed that take all automation specific factors into account. But such a simulation can only be performed in the late phases of the development when, for example, the code is available. To specify the electronics of an automation system, tools like EPLan [EPL17] (13) are used.

All of these approaches focus on a very specific aspect of the automation system and not on the system as a whole. However, there are also several approaches like SysML4AT [VHSFL14] (14), [SW09] (15), or [TF11] (16) which abstract

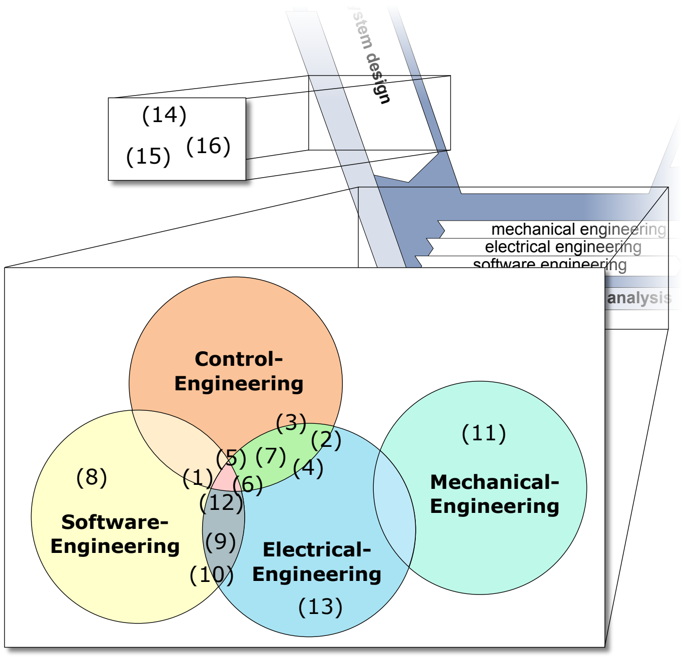


Figure 1.3: Disciplines and their specific analysis tools

from these discipline specific tools and provide information on Systems Engineering models or on a similar level of detail. These approaches are not intended to analyze the performance of a PLC in the early development stages.

They also differ in their level of detail and their notation towards the domain of industrial automation systems. Some of them provide influence factors and parameters in a well-know terminology and abstraction level used by automation system developers (e.g. [CHL$^+$03, LF07]). MARTE [Obj06], Palladio [RBB$^+$11], [Bon09] or [Wan06] only provide general means to model factors that influence the automation system. The developers must identify and decide themselves which factors must be regarded in an automation system and model them in detail.

Another problem is the integration of these prediction, respectively evaluation, approaches into the early stages of the development. They usually require in-depth knowledge of the system which is – at least to this level of detail – not available until the discipline specific development starts. If this knowledge is available, if assumptions are made, or if they are set as specific requirements, this information is usually solely persisted in the discipline-specific models and not propagated to the other disciplines.

Systems Engineering (SE) models provide a rough sketch of the system under development. Incorporating automation system-specific influence factors into Systems Engineering models is a necessary prerequisite to enable the early validation of the design models. SE approaches like CONSENS [GLL12, IKDN13] provide the possibility to model each sensor and actuator, fieldbus, and task allocation. But they lack the capabilities to specify program execution times as upper bounds for the developer to withhold, they lack a PLC independent way to model resource usage of a program, and they lack the capability to model properties on a suitable level of abstraction. By using additional profiles like MARTE [Obj06] this information could be added to the System Engineering models. However, this would lead to more complex models, reducing the overall usability and forcing the developers to use a detailed modeling language for embedded devices. An automation-domain-specific notation that enables developers to easily annotate existing Systems Engineering models is missing.

1.2 Contribution

Developers involved in the early phases of the development, respectively the system design phase, take different systems engineering roles [She96]. This thesis focuses on the role “System Analyst”, whose task it is to validate the overall system design with respect to the performance of the PLC. To do this, the two main problems that have been identified in the previous section need to be solved. First, it is unknown which influence factors must be considered when validating the selected PLC of an automation system in the early development stages. And second, formalisms and appropriate processes to capture these influence factors in Systems Engineering models are missing. This thesis provides the following contributions to solve these problems:

C1: Identification of Influence Factors
Several approaches exists that focus on the analysis of a specific part of an automation system. Each of these approaches also has a different view on the problem and therefore varying influence factors they cover. This means a developer can only model a network and its parameters in one tool and needs a different one to model the software services of an automation system. A complete list of influence factors that impact the overall automation system and which can be used to evaluate the design in the early stages of the development is not available. Therefore, the first contribution of this thesis is the identification of these influence factors.

An extensive list is gathered that details each influence factor in an automation system that impacts one or more quality of service properties. It is discussed whether the influence factors are available in the early development stages, what assumptions have to be made, its overall impact on the system, and which parameters need to be taken into account.

C2: Method for Modeling Automation System Influence Factors
Current System Engineering methods support the development of automation systems, but domain-specific performance-relevant information is neglected. It is possible to use performance modeling profiles like MARTE [Obj06] or SPTP [Obj05] to achieve these goals. However, they focus on the annotation of only

software relevant factors, are domain unspecific, and are tailored for a detailed, in-depth analysis of software functions. The second contribution of this thesis is therefore a method for capturing automation specific influence factors on a high level of abstraction that is applicable for the use in the early phases of the development.

For this thesis, a method has been developed that allows the automation system developers to specify influence factors in the early development phases of an automation system. This method consists of a formal model to capture the influence factors based on Systems Engineering models and a process to guide developers during the specification of these factors and parameters.

As an exemplary Systems Engineering approach, CONSENS has been selected. CONSENS provides means to model complex systems on a high abstraction level, providing developers of multiple disciplines the possibility to synchronize and coordinate themselves. It has been selected due to its rising use in the domain of mechanical engineering [TDBG15] and because a detailed model from a given real world application has been made available for this thesis (see Chapter 3). Additionally, in [HBM$^+$15, HBM$^+$16] a fitting development process has been proposed which could be extended to cope with the definition of influence factors with minimal effort.

In addition to these two contributions, a prototypical tool-suite has been developed. It allows developers to annotate existing CONSENS Systems Engineering models and to automatically derive inputs for a performance simulation. The simulation is used to predict quality of service attributes of the automation system under development. The scope of this thesis only covers a simulation of the overall utilization of a single PLC. The results of this simulation can be used to evaluate the current system design by providing predicted values of the PLCs utilization.

1.3 Overview

To achieve the stated contributions and to provide means for predicting the utilization of a PLC for a final evaluation, the following steps depicted in Figure 1.4 are carried out.

First (Step 1), this thesis deals with the identification of influence factors that will impact the overall PLC performance. To find the necessary factors, three sources will be considered: Exemplary Systems Engineering models of automation systems, industrial firmware of a well know PLC vendor (Phoenix Contact), and common factors from existing performance prediction approaches. For each influence factor, the respective parameters are identified. With regard to the performance prediction in the early stages of the development, the influence factors are either used, neglected, or simplified depending on whether it is necessary or appropriate to model them. **The result of this step is a list of influence factors and their parameters**.

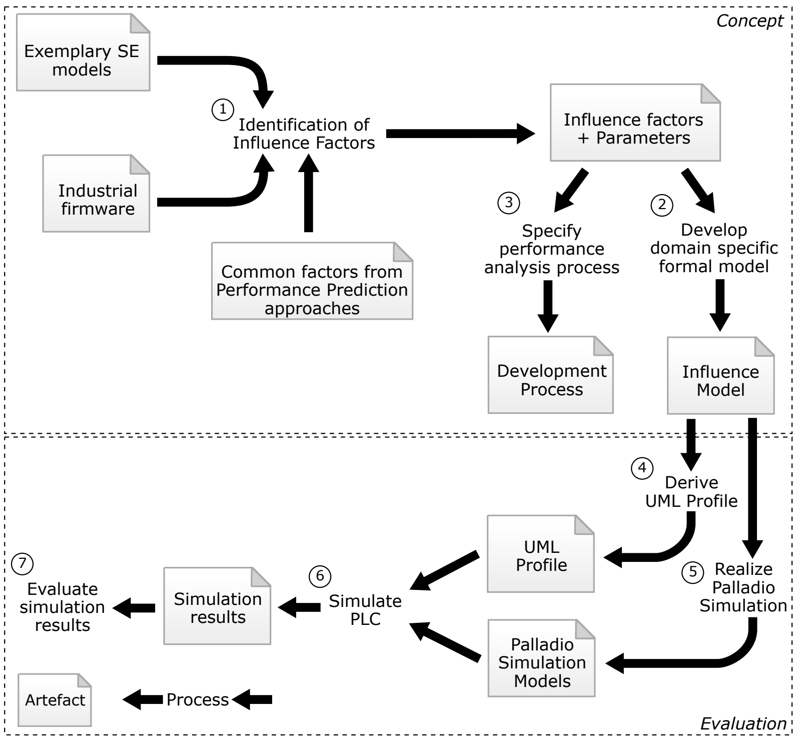

Concept
Exemplary SE models
Industrial firmware
1 Identification of Influence Factors
Common factors from Performance Prediction approaches
Influence factors + Parameters
3 Specify performance analysis process
2 Develop domain specific formal model
Development Process
Influence Model
4 Derive UML Profile
UML Profile
5 Realize Palladio Simulation
Palladio Simulation Models
6 Simulate PLC
Simulation results
7 Evaluate simulation results
Artefact
Process
Evaluation

Figure 1.4: Approach for identifying and developing an automation specific performance analysis based on System Engineering models

Second (Step 2), a formal model to capture the selected factors and parameters is developed. Some influence factors might be directly modeled, some must be derived from automation specific elements like sensors, actuators, fieldbusses, or provided services. The formal model incorporates domain specific elements and therefore allow developers to use their well known terminology. This approach improves the usability and prevents misunderstandings between the involved domains during the development of the automated system. The outcome of this step is a **formal model that captures the influence factors, their parameters, and relations to each other** in the context of a CONSENS Systems Engineering model.

Afterward, a process is defined based on the identified influence factors (Step 3). This **process** will guide developers through the **specification of factors and parameters** in the formal model. The process defines, at which point in the development process of an automation system a performance analysis should be carried out and what information needs to be gathered before starting the analysis. Basis for this development process is an existing CONSENS process developed for the specification of software requirements and their analysis in the early stages of the development.

To be able to evaluate the proposed method, a UML profile is specified (Step 4) that extends the SysML4CONSENS profile provided by [KDHM13, IKDN13]. The profile and models are created for and with the Papyrus Modeling environment [Ecl17c] (as part of the Eclipse IDE [Ecl17a]). The **UML profile captures selected influence factors and allows to annotate existing automation system models**. A running example introduced in Chapter 3 serves as a basis for these influence factor annotations.

Parallel to the creation of the UML profile, a suitable simulation tool is selected (Step 5). This tool and its models should be easy to use, provide a good interface to access or generate input models and support a fast and reliable performance simulation. Based on these and further requirements, the Palladio Component Model (PCM) [BKR09, RBB^{+}11] (see Chapter 2) has been chosen as the primary performance simulation tool/framework. This framework is usually used for the performance modeling and simulation of software architectures of high performance servers and clusters in the early design phases. For this thesis, the capabilities of Palladio are extended to support the automation domain specific requirements. To be able to simulate a PLC, the according Palladio models need to be created as well as specific load profiles captured from an existing PLC.

The UML profile and the Palladio simulation models can be used to simulate a PLC in the context of the modeled automation system (Step 6). Finally, the predicted utilization of the PLC is compared to measurements taken from a performance prototype which resembles the basic characteristics of the exemplary automation system (Step 7).

1.4 Thesis Outline

This thesis is structured as follows. In Chapter 2, the foundations necessary to understand the contents of this thesis are given. It includes an introduction into the automation domain and provides insights to the used modeling and simulation tools. An exemplary automation system is used throughout this thesis. An overview of this system as well as selected, detailed CONSENS models are provided in Chapter 3. Chapter 4 gives an overview of the identified influence factors that impact the performance of a PLC or automation system. Each factor is analyzed and a decision is made, whether it should be modeled in the early development phases and to what degree. The following Chapter 5 introduces the proposed development process and lists related work. The second part of this chapter covers the formal aspects of modeling the influence factors. Chapter 6 details the UML profile that is used to model the example automation system and its influence factors. This chapter also covers the selection of a fitting prediction approach and the follow-up design of the Palladio models. Therefore, an excerpt of the simulation models used to conduct a performance analysis for a PLC are introduced. Also, several semi- and fully automatic transformation steps that reduce the need for human interaction when deriving the simulation from the UML-based Systems Engineering models are briefly explained. To evaluate the developed concepts, Chapter 7 sets up the evaluation context and details the setup of the performance prototype. Afterward, the simulation results are compared to a performance prototype. The results of this comparison will be discussed and threats to validity of this evaluation are pointed out. Finally, in Chapter 8, a short summary is given and open questions and future work are discussed.

CHAPTER 2

Foundations

This chapter introduces the foundations for this thesis. The first Section 2.1 will give a brief introduction into the automation domain. Important concepts and essential components which impact the design of an automation system are highlighted. In Section 2.2, a short introduction to Systems Engineering is given. This section also includes the presentation of CONSENS and its models, the visual modeling language SysML as well as the SysML4CONSENS profile which is used to create CONSENS models based on SysML. The SysML4CONSENS profile is used to create a formal, analyzable CONSENS model (see Chapter 3) and is a required foundation for the influence profile presented in Chapter 6. The last Section 2.3 focuses on the area of performance modeling and prediction. In Subsection 2.3.1, the MARTE UML2 profile will be presented which allows annotate existing models with performance relevant annotations. Subsection 2.3.2 introduces the Palladio Component Framework, its models, and various analysis tools to predict the performance of a system.

2.1 Industrial Automation

The idea to reduce the need for human work in the production of goods and services by automating tasks exists for a while now. A classic example of an automated task is the grinding in water- or windmills, in particular, the part of turning the millstones. This automation of tasks increases the productivity and replaces animal or human workforce. Today, complex tasks are performed by a combination of electrical and mechanical machines like conveyor belts, drills, or robots and the domain is commonly described as automation or industrial automation [Lau13, FMS04, GG13, Lun08, KRK95].

Most automation tasks are hierarchically structured, leaving the coordination tasks on higher levels and critical real-time tasks closer the hardware. They are controlled by Programmable Logic Controllers (PLC), running programs written in one of the languages specified in the IEC standard 61131-3[Int13a]. The automation tasks are performed by a combination of electrical and mechanical machines like conveyor-belts, roller, pusher, drills or whole robots. A PLC uses inputs gathered by sensors and decides upon these information how to control the actuators. Examples for sensors are light barriers, heat or pressure sensors. Typical actuators in a production environment could be motors in conveyor belts, valves or hydraulic pushers. In the early days of PLC based automation, these sensors and actuators – also named IO for input/output – were connected directly to the PLC. Later on, due to the high amount of cables and the resulting installation difficulties, fieldbusses were used to send grouped or even compressed data between the IOs and the PLC. PLCs are used for a various range of tasks in different domains like factory automation, waste water treatment, traffic control, lifts, public transportation or modern energy networks.

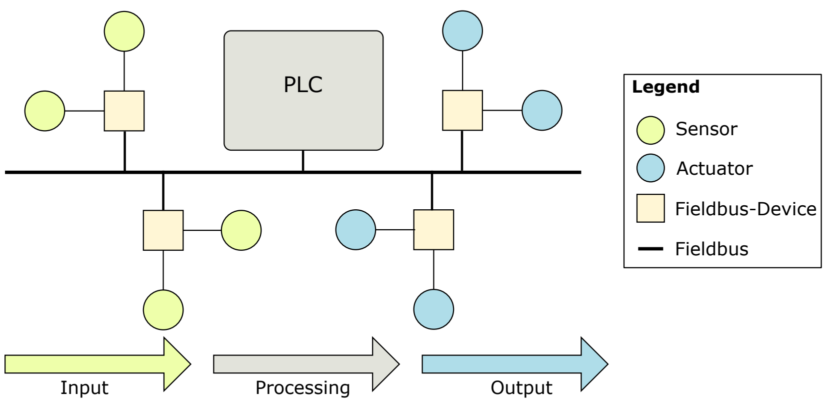


Figure 2.1: Schematic representation of a simple automation system

A schematic of a small automation system is given in Figure 2.1. The PLC in the center queries sensor data of the system it controls. Based on the current state of the PLC and the inputs from the sensors, the PLC decides what commands or values are send to the actuators. In general, the shorter the time of these repeating steps (small sampling rate) is, the more accurate the system can function. Small sampling rates are, for example, needed if a conveyor belt moves goods that needs to be sorted. A sensor identifies a good, an actuators pushes it from the belt. If the time between the identification and the signal to the pusher takes to long, the good has already passed the position. However, smaller cycle times lead to more calculations and a higher utilization of the PLC. In general, it is the goal to use cycle times as small as possible. This will increase the precision of the automation system and also allows to move goods faster and, therefore, increases the overall throughput of the system.

The PLC follows a strict order of actions to execute instructions for the automation task (see Figure 2.2) in such a cycle. After powering on or a reset the PLC first initializes all values for the programs. This includes all variables, counters, markers as well as all used input data provided by the sensors. Afterward, a loop starts with three steps. First, all inputs are read from the IO system (e.g. a fieldbus controller). Then the different instructions are executed and commands for the actuators calculated. After the last instruction execution, the local data is copied from memory to the output image where a fieldbus controller send it directly to the actuators.

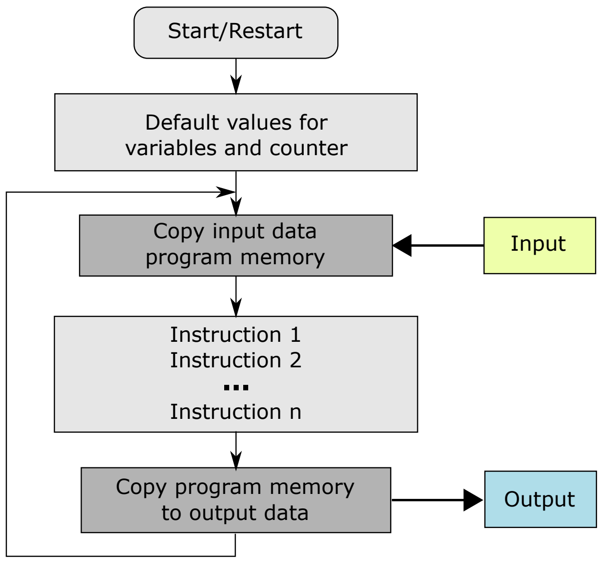

Start/Restart
Default values for variables and counter
Copy input data program memory
Input
Instruction 1
Instruction 2
...
Instruction n
Copy program memory to output data
Output

Figure 2.2: Different steps of the of the PLCs cycle

2.1.1 IEC 61131-3

Soon after the industrial automation was becoming more and more important for modern production systems, the need to standardize the programming of PLC was inevitable. Today, the de-facto standard for programming PLCs is the IEC 61131-3 standard [Int13a] specified by the International Electrotechnical Commission (IEC) [Int16a]. The standard is composed of nine parts each defining different aspects of automation systems and components. The latest version has been released in 2013. For programming PLCs, the interesting part of the standard is "Part 3". It specifies the syntax and semantics of a set of five programming languages. The third edition of this standard introduced a range of new features like additional data types, conversion functions, references, name spaces and the most importantly object oriented features of classes and function blocks. The following subsections give a brief introduction to the common parts of the standard and these different programming languages.

All of the languages share IEC 61131 common elements. The following list is giving a short overview over some of these elements. A complete overview of the core elements as well as additional constructs can be found in the IEC 61131-3 standard or in [JT10].

- **Configuration**: The configuration is the main container for global program variables and settings. In most PLCs the configuration is the equivalent to a runtime environment. Therefore, the term IEC runtime environment covers/includes the configuration in this thesis. The IEC runtime environment provides an execution context, data from fieldbusses, global variables, and allows the exchange of data between programs.
- **Resource**: The resource represents a (logical) processing unit on which the programs are executed. In the early days of industrial automation,

each resource was a single CPU and most of the PLCs contained only one. Due to the increasing use of multicore systems, a PLC may contain more than one resource, but this can be rarely found.

- **POU**: The Program Organization Unit is a generic name for Functions (ADD, SQRT, SIN, COS, GT, MIN, MAX, AND, OR, etc.), Function Blocks or Programs
- **Program**: Programs are the top level elements which can be used to structure the application. Each Program is written with one of the five IEC languages and executed via tasks. A Program can contain Function or Function Block calls, but cannot execute other Programs.
- **Function Block**: The IEC 61131-3 Function Blocks (FB) can be compared to programs. They must be executed within a task or program and – in contrast to functions - keep their internal variables after their execution. A Function Block can have both input and output parameters and must be instantiated from a previously defined type.
- **Function**: A Function must be declared and can afterwards be called from Programs, Function Blocks, and other Functions. They do not persist any variables.
- **Task**: This element is used to trigger the execution of Programs and Function Blocks. There are different execution behaviors for tasks (cyclic, event based, and idle) which are explained in more detail in Chapter 4. Tasks have priorities and can preempt each other.

Figure 2.3 summarizes the different elements of the IEC 61131-3 standard. A Configuration can contain several Resources. Each Resource triggers various kinds of Tasks. Each Task contains a Program, respectively a Program instance. A Program can further use instances of Function Blocks and Function calls to hierarchically structure the functionality of the Program. In the following

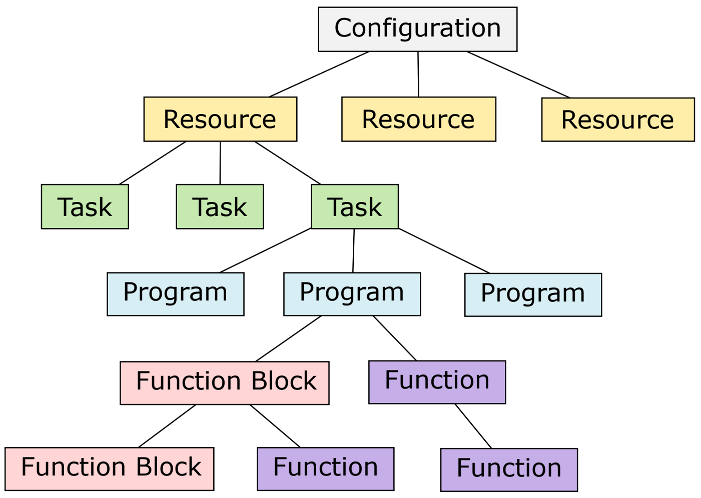


Figure 2.3: Structure of the IEC core elements

subsections, the different IEC 61131-3 programming languages are listed with simple (code) examples and a short description.

Instruction List (IL)

The Instruction List is a low-level machine-oriented language offered by most of the programming systems. It can be compared to the assembler language. Its main advantage are the detailed, easy to trace instruction steps which help debugging machines with a lot of sensor/actuator interaction. However, it is not recommended to use this language for more complex programs and algorithms. The listing below shows an excerpt of a training exercise from the Beckhoff Automation website [Bec17].

Listing 2.1: Instruction List code example (source [Bec17])

```
LD      TRUE    (*load TRUE in the accumulator*)
ANDN    BOOL1   (*execute AND with the negated value of the BOOL1 var*)
JMPC    label   (*if result TRUE, then jump to the label "label"*)
LDN     BOOL2   (*save the negated value of *)
ST      ERG     (*BOOL2 in ERG*)
label:
LD       BOOL2          (*save the value of *)
ST       ERG            (*BOOL2 in ERG*)
```

Ladder Diagram (LD)

The Ladder Diagram offers a more electronic influenced approach to program PLCs. The structure and elements in this language resemble ladders based on the circuit diagrams of relay logic hardware. They are widely used where sequential control of a process or manufacturing operation is required. Figure 2.4 shows an example taken from the Beckhoff Automation website [Bec17]. Like an electrical can the ladder digram be read from left to right. Elements in brackets represent contacts (sensors) and in round parenthesis actuators. Depending on closed brackets, a path can be traced from the left to the right side. Like in electrical plans, this would lead to an activation of all actuators on this path. More information about Ladder Diagrams can be found in [JT10].

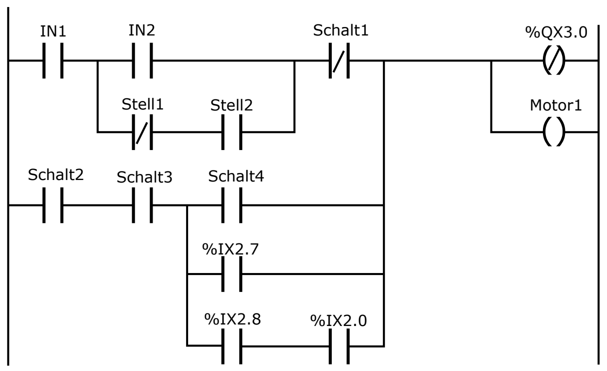


Figure 2.4: Example for a Ladder Diagram code snipplet (source [Bec17])

Structured Text (ST)

Structured Text can be best compared to the Pascal programming language. It is the best IEC language for programming algorithms and complex behavior in automation systems. It offers predetermined structures for often used constructs such as IF, CASE, FOR, WHILE and so on. The code shown in listing 2.2 specifies variables that are used as state for a state machine.

Listing 2.2: A code snipplet written in Structured Text (source [Bec17])

```
init := 0;
enabled := 1;
working := 2;
disabled := 3;
ready := 4;
if state = init then
  if st_tcp_ready and st_sql_ready then
    state := enabled;
  end_if;
  RETURN;
end_if;
```

Function Block Diagram (FBD)

Another common graphical language is the Function Block Diagram (FBD). Each Function Block (FB) must be specified and then instantiated in a Program or parent FB (similar to classes and objects). They hold their internal (state) variables as long as the PLC runs and provide invars, outvars and inoutvars as ports to exchange data with each other. The Figure 2.5 shows a simple example of multiple Function Blocks that are connected with each other. There are basic FB specified by the standard like MUL, ADD or AND which can be used to compose new Function Blocks. It is also possible to specify the internal behavior of a Function Block with one of the other languages. The example diagram is also taken from the Beckhoff website [Bec17]. Function Blocks allow the structuring and modularization of complex automation systems. One or more Function Blocks can encapsulate functionality of a hardware module or allow the reuse in different applications. The latest version of the IEC 61131-3 introduced inheritance to Function Blocks, focusing even more on object oriented design of complex automation systems and enabling better implementation of product lines. More on Function Blocks and their specific execution behavior in 4.3.2.

Sequential Function Chart (SFC)

This language is not specified directly in the IEC standard, but in its referenced document IEC 60848: 2002, GRAFCET Specification language for sequential function charts. It has been originally developed by SIEMENS and is now adapted by many programming tools and PLC vendors. SFCs break a sequential task down into steps, transitions and actions. Therefore is this language ideal

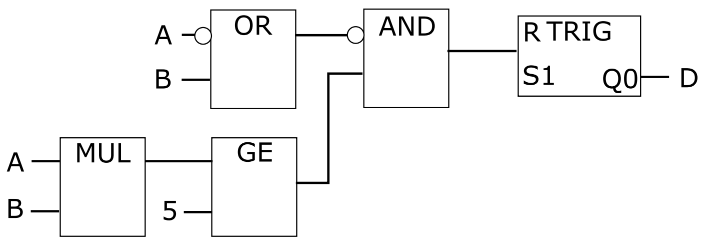


Figure 2.5: Example of a Function Block Diagram (source [Bec17])

to program and visualize sequences of actions the PLC must execute. SFCs also provides constructs for branching or parallel execution. Figure 2.6 shows three states and transitions between them. In state Filling ready the pump will be turned on.

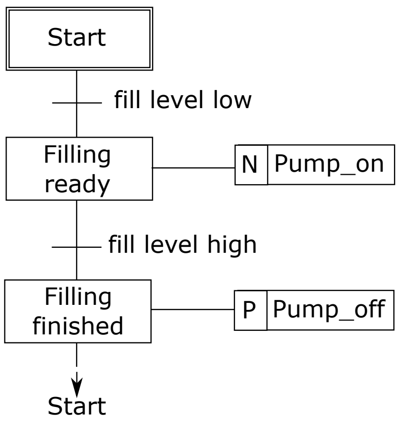


Figure 2.6: Example Sequential Function Chart (source [Bec17])

2.1.2 Fieldbusses and automation networks

In the beginning of industrial automation, each sensor and actuator (IO) was directly connected to the PLC. This lead to very complicated wiring and in some cases also to severe bottlenecks in space to route the cables. The need to combine different IOs in just one connection became obvious and therefore multiple IOs needed to be multiplexed in one signal cable. This is done with a bus system. In industrial automation, such a bus system is also called field bus, due to its goal to connect the different IOs in the field with the PLC in a wiring cabinet [SW08, G^{+}01]. Today, several hundreds of bus systems exists, each with different properties regarding speed, topology, bandwidth, delay and more [Tho05, LaÎ99, Mah13, GJF13, GH13, PN09]. In 1999, the IEC standards committee created the initial form of the IEC 61158 standard [Int03] with eight different protocol sets to fix the foundations for the most common fieldbusses. These sets included, among others, PROFIBUS and Interbus which will be briefly introduced in the following Chapters.

Figure 2.7 shows a conceptual structure of an automation system. In the upper part is the control network or office network. Supervisory systems, engineering workstations and control station are placed here to access data in the subnetworks below. On the left side is a PLC with IOs connected via bus couplers. These bus couplers are used to multiplex and demultiplex the different values from and to the sensors and actuators. Depending on the fieldbus technology they are also called slaves, devices and so on. The bus couplers contain one or more modules to connect the IOs. In the subnetwork on the right side, the bus couplers are connected not directly to the PLC, but are part of the control network. This kind of topology is becoming more and more popular due to the use of industrial Ethernet [DSA$^+$14]. They simplify the integration due to the common Ethernet and the overlaying IP protocols. Another trend is the use of wireless communication [MLK06]. Omitting cables for example, allows the design of more flexible production facilities or an easy exchange of tools on a robot. The PLC in the figure shows a PLC with three buscouplers connected via a wireless network.

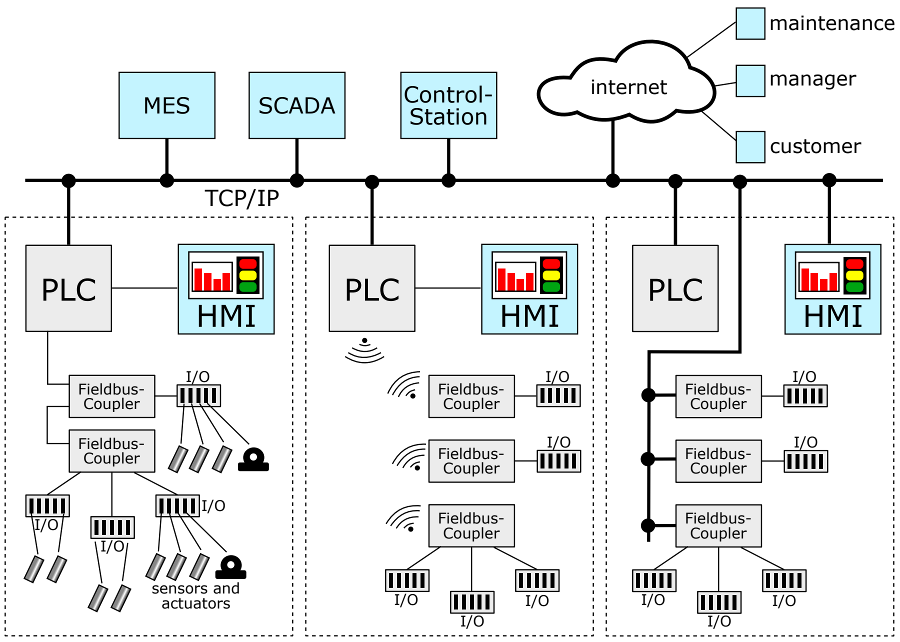


Figure 2.7: Conceptual fieldbus structure of an automation system

The different kinds of fieldbus technologies each have their specific design goals. In general, the primary goal is to exchange data between the PLC and the sensors/actuators in a deterministic and fast way. Standard communication usually requires a response of 100ms. Factory automation requires a response time in the order of 10ms. And for precise control of actuators for motion control, response times around 1ms are necessary with a jitter below 1 µs. Some more details on fieldbusses and their specific properties can be found in Chapter 4.

In general, the output of a sensor or input of an actuator can be a digital or an analog signal. When using a digital fieldbus, an analog value in form of a certain voltage must be converted to a digital value with an AD-Converter. Digital sensors or actuators usually include such an AD-Converter. In the context of fieldbusses, the term digital or analog is also used as the type of IO device, with respect to its data size that is transmitted via the fieldbus to the PLC. A light barrier that only sends a signal when an object is detected is an example for a digital IO. The data this sensor transmits can be represented with just 1 bit. Analog devices can provide more values in a set rage by using a bigger data size. For example, can this range depend on the voltage the sensor outputs. This voltage is converted to a digital signal. A temperature sensor may provide a specific voltage for its given temperature range which is then converted to an 8 bit value. Providing a greater data size (e.g. 16 bit) will result a finer resolution or a greater range of values.

2.1.3 Topologies and network devices

A network topology describes the structure of a network, including its nodes and connections. In a fieldbus, nodes usually are the PLC, fieldbus devices and IOs. The PLC often takes the role of the master controlling each slave. The fieldbus device can be roughly described as a device that multiplexes/demultiplex the IO data provided by the sensors and actuators. There are two ways of defining a network structure: the physical topology and the logical (or signal) topology. Physical topology is the concrete layout of the network as realized with IOs, devices, cables, and via other connections like wireless setups. The logical topology refers to the nature of the paths the signals or messages are send from node to node. Some fieldbusses can even be realized by different physical topologies. More information on fieldbusses and their topologies can be found in [Tho05, G^{+}01]. A comprehensive overview and in-depth introduction to computer networks is provided by Tanenbaum [Tan02].

Figure 2.8 shows the common topologies. In a Ring (a) all devices are connected to a predecessor and a successor. All data transmitted between nodes travels from one node to the next node in a circular manner. Dual-Rings also provide a second connection backwards which for example allows the network to function even if the connection between two nodes is separated. In the bus network topology (b), every node is connected to a main cable called the bus. Each node is therefore directly connected to one another. All data that is transmitted between nodes is able to be received by all nodes. The tree (c) has a “root” node at the top level of the hierarchy. Each node has further connections to its child nodes, but not to nodes on the same hierarchal level. A message or signal from the root the one of the leafs has to be forwarded by all nodes in between. The Star topology (d) has a single node in the center and from this node connections to all remaining nodes in the network. No other connections between nodes are allowed. The Line (e) has a starting and an end point with multiple nodes in between. All node between the start and end points are connected to a successor and a predecessor. The Meshed (f) and fully Meshed (g) networks contain nodes with a point-to-point link. Messages to a not directly connected node must be forwarded by nodes in between. In a fully Meshed network, data can be simultaneously transmitted from any single node to all of the other nodes.

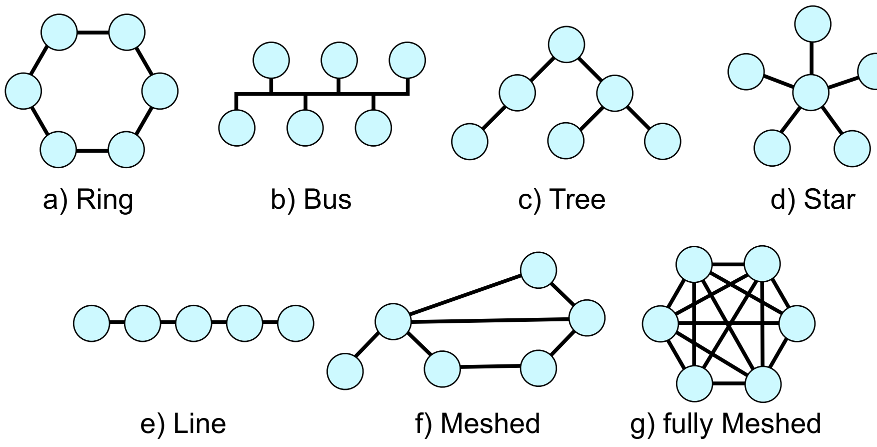


Figure 2.8: Examples of (fieldbus) network topologies

2.1.4 Automation pyramid and management systems

Automation systems currently consists of multiple layers, each for a different purpose and very specific requirements. Figure 2.9 shows the Automation Pyramid [FMS04, KRK95] which is often used to explain these layers. At the bottom of this pyramid are the already introduced sensors and actuators. The data provided by the sensors is used by the PLCs on the next level to calculate the output used for the actuators. The communication on this level is in real-time and has very strict requirements to performance and determinism. Multiple PLCs are often coordinated by a SCADA system which itself is controlled by an Manufacturing Execution System. These systems are briefly explained in the following sections. A similar classification into levels is provided by the IEC 62264 standard [Int13b].

SCADA
Supervisory Control and Data Acquisition (SCADA) systems are used in automated systems to monitor and control technical processes. These systems collect and evaluate process data from multiple PLCs or directly from their IOs. SCADA systems can also manipulate data and therefore be used to coordinate, respectively control multiple PLCs in larger automation systems like huge chemical processing plants. In the first generation of SCADA systems (monolithic), central mainframes controlled the PLCs. For this purpose, dedicated communication links and instructions send with proprietary protocols were used. The second generation (distributed) made use of local area networks to send and receive data from different PLCs in real time. Each PLC

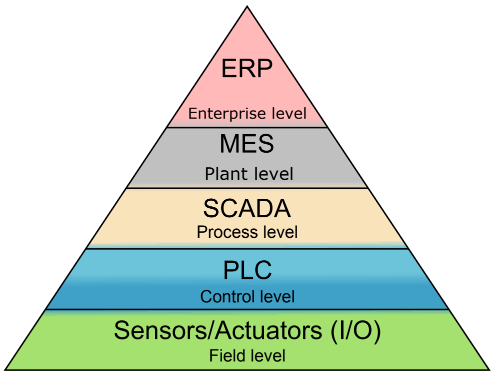


Figure 2.9: The automation pyramid with different function levels (based on [FMS04, KRK95])

was responsible for a specific task, which made the monolithic approach obsolete. In the last generation (networked), a continuous network between PLCs, services at the management level (see ERP and MES), and to other external systems via the Internet is realized. This allows the exchange of data between any systems with ease. Communication between SCADA systems, PLCs and other automation devices is realized by different protocols. OPC (OLE for Process Control) and its successor OPC-UA (Unified Architecture) are widespread protocols based on TCP/IP.

Manufacturing-Execution-System (MES)
The Manufacturing Execution System (MES) is directly connected to the control technology and serves as an intermediate layer to the lower levels (SCADA & PLC & IO) and the above lying resource planning (ERP). The tasks of MES includes enterprise data acquisition, machine data acquisition and personal data collection. The MES is mainly used for continuously controlling the enforcement of an existing production planning and the feedback from the process. They identify for example material shortages and report them to the upper levels. It is a trend that MES access PLCs and their data directly, using the standardized interfaces such as OPC-UA.

Enterprise-Resource-Planning (ERP)
ERP systems are software systems that are used for detailed resource planning in a company. Well-known ERP systems are for example SAP ERP from SAP [KG17] or Oracle's E-Business Suite [Ora17]. ERP systems reflect the business processes of the company and map them to the underlying tasks for PLCs. They provide important information used mainly in materials management, production, finance and accounting in an enterprise. Often ERP systems have a rich set of different interfaces to other (external and internal) systems such as web shops or other business-2-business platforms. The data ERP systems need is usually provided from the underlying MES system.

2.2 Systems Engineering

"An interdisciplinary approach and means to enable the realization of successful systems" is the short definition of Systems Engineering by the International Council of Systems Engineering (INCOSE) handbook [WRF$^+$15]. At the core, Systems Engineering focuses on how to develop and manage complex systems over their life cycles with a holistic view onto the system under development. Using a Systems Engineering approach for the development of complex systems should increase the overall effectiveness of projects. By front loading effort and investing in communication and synchronization, time- and cost intensive errors in the late development phases can be reduced [EGEE$^+$08]. This includes, among other points, the analysis of customer needs, capturing functional and non-functional requirements, design, and system validation. It is a interdisciplinary field of engineering and management that, for the domain of automation systems, usually involves three disciplines: Mechanical engineering, electrical engineering, and software engineering. Over the past years, the traditional document-based systems engineering is slowly replaced by the Model-Based Systems Engineering (MBSE) as the future development paradigm for technical systems [BOF$^+$14]. MBSE approaches make use of a formal system model that describes elements and relations between them, and ensures a common basis throughout the development phase for all involved disciplines [RBG13].

Depending on the used approach and the formality of the underlying models, different kinds of subsequent or even parallel analysis can be conducted [BFF90, ZPK00, Smi62, PJ08, MU09, Lop15, RBG13]. These analysis help to validate the system design in the early development stages. This will reduce costly time- and cost intensive changes in the integration phase of a project.

To create such Systems Engineering models, different methodologies are available, each using varying processes, capability models, and languages [E$^+$07, E$^+$07]. In the following subsections, CONSENS (CONceptual design Specification technique for the ENgineering of complex Systems) and SysML, as a visual modeling languages, will be briefly described. Other well known Systems Engineering approaches are for example, SYSMOD [Wei15], Telelogic Harmony-SE [Dou06, Hof08], Object-Oriented Systems Engineering Method (OOSEM) [LFM00], Rational Unified Process for Systems Engineering (RUP SE) [CP03], or Dori Object-Process Methodology (OPM) [Dor11].

2.2.1 CONSENS

The specification technique CONSENS (Conceptual Design Specification Technique for the Engineering of Complex Systems) is a Systems Engineering approach to specify the product and its according production system [GDKN11, DDGI14, GFDK09, Fra06, GLL12]. In CONSENS, the developers create a general system model, specifying (among others) requirements, functions and an abstract structure of the system to develop. If this system model, named *principle solution*, has reached a mature state, the different disciplines start with their discipline-specific realization. Changes which occur in these phases can be synchronized with the other disciplines, by using the principle solution [Rie14]. This helps to synchronize and coordinate the developers during the separated development.

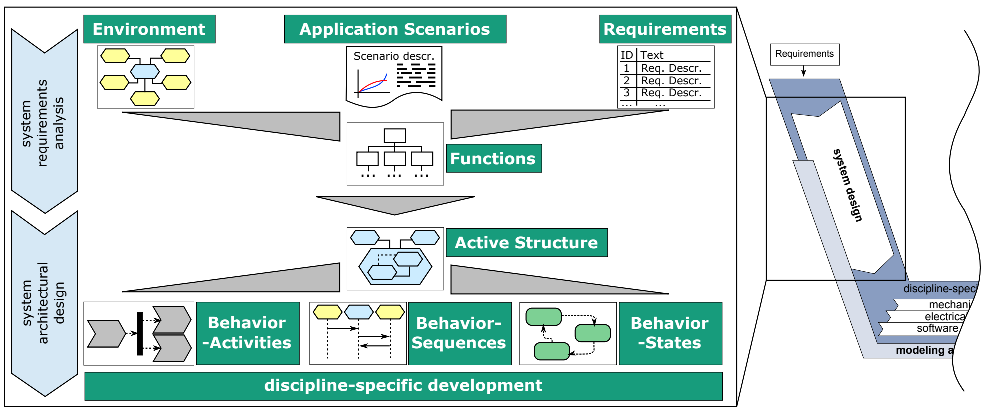


Figure 2.10: CONSENS partial models and alignment to the VDI 2206 process

Figure 2.10 shows the different partial models that form the principle solution and their order in the proposed development process. This process is part of/-based on the system design phase of the Vee-Model defined in the VDI guideline 2206 [Ver04]. These models and steps are described from top to bottom. During the system requirements analysis, three different models are created.

Application scenarios each describe a specific situation of the system. This includes the behavior of the system, the events that trigger this state and other descriptions to understand the system. The primary task of application scenarios is to describe a problem (for a certain situation) and a rough solution.

Another model of the principle solution covers the **Requirements** of the overall system, including the product as well as the production system. The list of requirements defines functional and non-functional requirements. They describe the desired functionality, behavior as well as properties of the system. The requirements are textually described and enriched by attributes and their characteristics.

An important part of the analysis is the **Environment** model of the system that has to be developed. The environment model focuses on elements that interact or influence the system under development and captures the relations between them. Examples for such interactions are SCADA systems that coordinate multiple production plants, logging server that access data, mobile devices, or humans using the integrated HMI to control the system.

The **Functions** model is used to create a hierarchical subdivision of the functionality, organized in as a tree. A function represents a general and required coherence between input and output parameters to fulfill a task. In CONSENS, functions are realized by solution patterns and their realizations. The solution patterns define the characteristics of an element that need to be realized as well as interactions between elements.

After the four models have been created, the system architectural design begins. The first step is the specification of an **Active Structure** model. It is used to give a hierarchical overview of the system structure by modeling system elements, their attributes, and relations between them. Different kinds of flows are used to further detail these relations between elements and how they influence each other. They are unidirectional or bidirectional and can be used to model energy, information, and material flows in the system. Figure 2.11 shows a simplified active structure diagram specifying system elements and flows in a production system. The *HMI* system element and the *SCADA* environment element communicating with the *MillingCenter*. For this, they use *Informationflows* (dashed lines). The HMI sends and receives control messages from the PLC, which is a sub element of the MillingCenter. Ports are used to type the kind of *Informationflow*. The same principle applies to the *Materialflow* (double lines) from the *Cooling Liquid Supply* to the *Mill*. The *MaterialFlow* denotes that the Cooling Liquid Supply element provides a cooling liquid to the mill. Via additional *Informationflows*, various commands and information messages are exchanged (e.g. current_pressure or status). The *EnergySupply* system element provides electrical energy via *ElectricalFlows*. CONSENS spe-

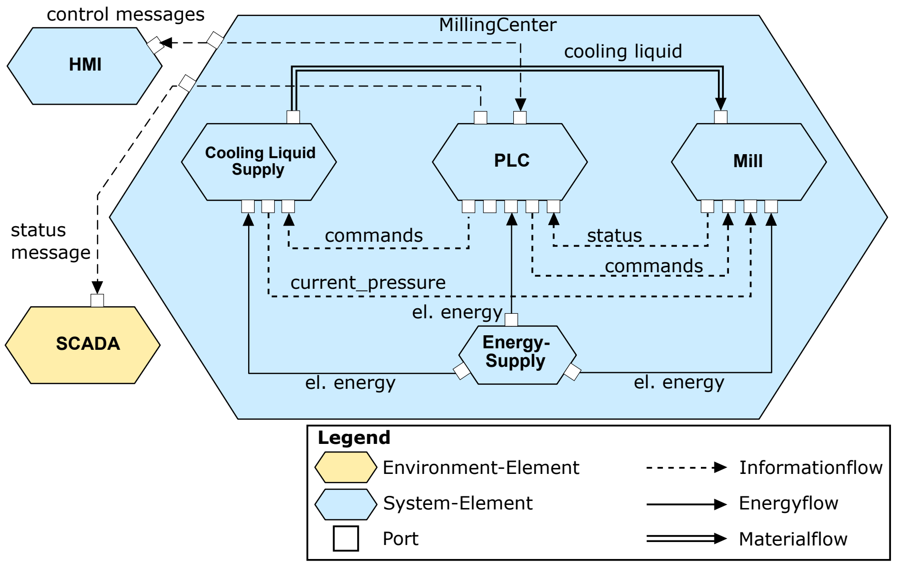


Figure 2.11: Exemplary active structure diagram

cifies additional elements to model for example disturbances, measurements, logical units, optional, or variant elements.

To further detail the behavior of the system (on a high level of abstraction), different kinds of behavior models can be used. Figure 2.10 lists **Behavior-Activities**, **Behavior-Sequences** and **Behavior-States**. Each provides means to model the discreet behavior of the whole or parts of the system. To create CONSENS models, the SysML4CONSENS UML profile [KDHM13, IKDN13] has been developed (see Chapter 6). It is based on UML, respectively SysML, and therefore makes use of UML Sequence-, Activity-, and State-Diagrams.

After the principle solution has reached a mature state, the system architectural design phase ends and the discipline specific development begins. During this phase, the disciplines can be coordinated, respectively synchronized, via the principle solution which holds system elements from each discipline. Further details of the process are described in [HBM+16, HSST13]. In addition to the introduced models, CONSENS provides means to specify the abstract manufacturing process sequences, the shape of a production system, the resources needed to manufacture the product, and a System of Objectives to formally represent external, inherent, and internal objectives and their connections.

2.2.2 SysML

The Systems Modeling Language, short SysML, is a standard provided by the OMG [Obj15b]. It is a graphical modeling language for specifying, designing, and verifying complex systems. SysML provides means to model for example hardware, software, information, procedures, requirements, and relations between these elements. A good overview and more in-depth explanations of the SysML modeling language can be found in the books written by Weilkiens [Wei11] and Friedenthal [FMS14]. SysML is the foundation of the SysML4CONSENS profile described Section 2.2.3.

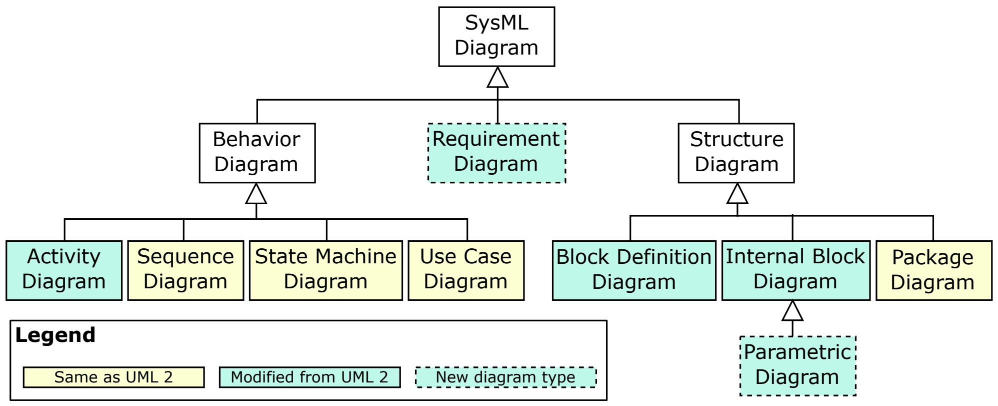
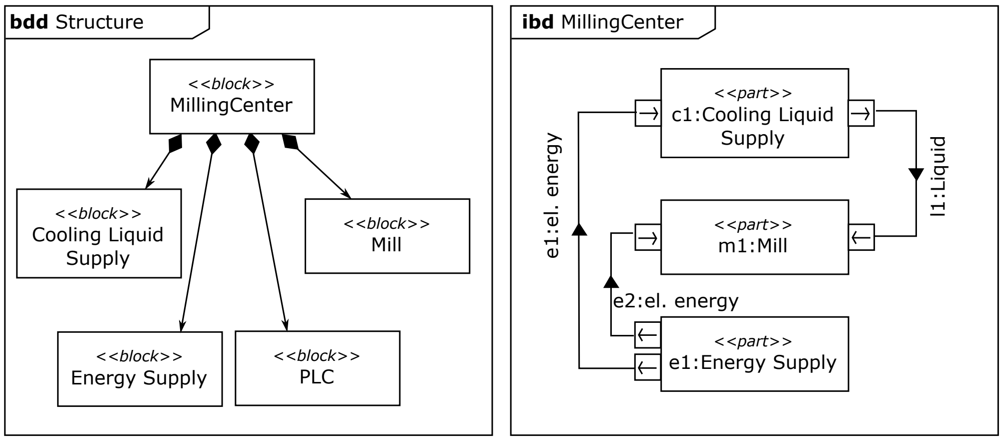


Figure 2.12: SysML Diagram Types [Wei11]

Figure 2.12 shows the diagrams that are used to specify different aspects of a model. Since SysML reuses a large part of the UML, existing diagrams can be reused and do not need to be modified/extended. SysML also provides new diagrams for various purposes and extends selected ones.

Core part of the SysML are the block definition and internal block diagrams that are used to model the structure of a system. A block definition diagram (BDD) describes the system hierarchy and components. The internal block diagram (IBD) is used to specify the internal structure of a system in terms of its parts, ports, and connectors. Similar to the UML is the package diagram used to organize the model. Figure 2.13 shows a simple example for the use of an BDD (left) and the detailing IBD (right). The BDD defines the types *MillingCenter*, *EnergySupply*, *PLC*, and *CoolingLiquidSupply* with their relations and properties. In the IBD, the internal structure of the MillingCenter is further detailed and parts (representing the instances) are connected via ports and flows. These ports specify interaction points on blocks and parts. Flow ports define what can flow in or out and are also types by a block, value type, or an explicit flow specification. More details on blocks, parts, ports and flows, a well as a similar example can be found in [FMS08].



(a) Block Definition Diagram (b) Internal Block Diagram

Figure 2.13: Exemplary BDD and IBD diagrams

To model the behavior of a system, four types of diagrams can be used that are imported from the UML specification. States, transitions, and actions are specified with the state machine diagram. Use cases describe functionality and interactions with other systems, system parts or users. Sequence diagrams provide an ordered view on the interaction between different parts of the system and the activity diagram represents the flow of data and control between activities.

Furthermore, SysML provides a requirement diagram to create a hierarchy of requirements. These requirements can be linked to various model elements and identify satisfy or verify relations between them. Constraints on system property values can be modeled with the parametric diagram. Such properties could be for example performance, reliability, length, or mass of an element.

SysML provides a visual modeling language and not a methodology to develop systems. However, SysML is used in broad range of Systems Engineering approaches [JLS11, BKFVH14, Alt12, IKDN13, VHSFL14, KVH13] including Rational Unified Process for Systems Engineering (RUP SE) [CP03], the Object-Oriented Systems Engineering Method (OOSEM) [LFM00], and Telelogic Harmony-SE [Dou06, Hof08].

2.2.3 SysML4CONSENS

The SysML4CCONSENS [KDHM13, IKDN13] is a UML profile that further extends the SysML language specification by CONSENS specific elements. The profile has been developed by Holtmann et al. to create formal models based on the CONSENS methodology. The SysML already provides a rich basis of elements that overlap with CONSENS to a high degree. This includes requirements, flows, activities, state machines, function hierarchies, and block diagrams.

Figure 2.14 shows an excerpt of the profile definition. The three kinds of Flows in CONSENS are *InformationFlow*, *EngergyFlow*, and *MaterialFlow* which can be easily represented by extending the SysML *FlowSpecification*. This approach will preserve the ability to use ports in the models and according editors. To model system elements, the *Block* is simply extended by the *SystemElementTemplate* stereotype. This also covers the environment model including the *EnvironmentElements* and *SystemTemplate*. More detailed examples of CONSENS models created with the SysML4CONSENS profiles are given throughout this thesis, starting in Chapter 3.

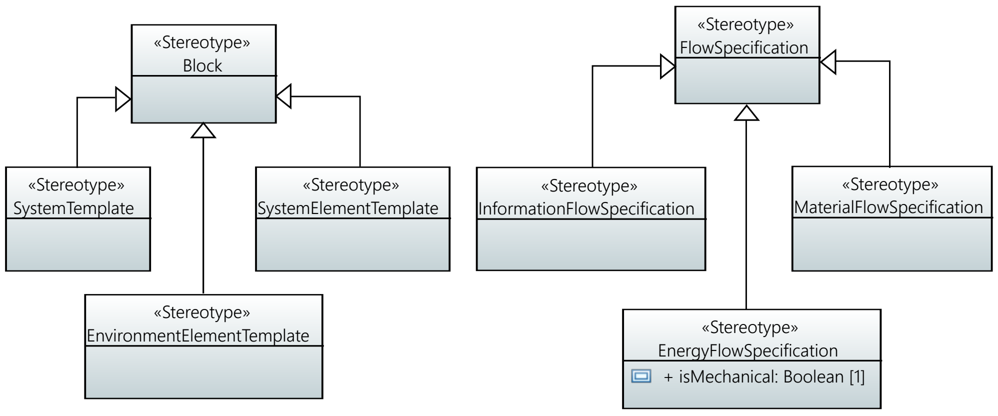


Figure 2.14: Excerpt of the SysML4CONSENS UML profile

The SysML4CONSENS profile is the basis for the Automation Influence Model (AIM) profile which extends or references a small subset of stereotypes. This profile is detailed in Chapter 6. Table 2.1 list all stereotypes defined in the profile as well as their corresponding base elements the stereotype extends. This base element is either a stereotype defined in the SysML 1.3 profile or a metaclass in the UML metamodel.

Table 2.1: List of used System and Environment elements

Stereotype	**Extends**
SystemTemplate	SysML1.3::Block
SystemElementTemplate	SysML1.3::Block
EnvironmentElementTemplate	SysML1.3::Block

Table 2.1: List of used System and Environment elements

Stereotype	**Extends**
InformationFlowSpecification	SysML1.3::FlowSpecification
EnergyFlowSpecification	SysML1.3::FlowSpecification
MaterialFlowSpecification	SysML1.3::FlowSpecification
ContinuousInformationFlowSpecification	InformationFlowSpecification
DiscreteInformationFlowSpecification	InformationFlowSpecification
MeasurementPoint	UML::Port
SystemElementExemplar	UML::Property
SystemExemplar	UML::Property
EnvironmentElementExemplar	UML::Property
LogicalRelation	UML::Connector
Function	UML::Class
FunctionContainment	UML::Association
Induce	UML::Abstraction
ApplicationScenario	UML::UseCase
partialModel	UML::Package
MechanicalConnection	UML::Connector
LogicalGroup	UML::Comment
FlowConnector	UML::Connector
ConsensRequirement	SysML::Requirement

Since the base concept of CONSENS does not provide types and instances, an equivalent mapping to SysML and UML must be made. The stereotypes *SystemElementTemplate* and *EnvironmentElementTemplate* extend the SysML *Block* to specify a type. Such a type can be referenced by its instance, specified by the *SystemElementExemplar* and *EnvironmentElementExemplar* which are based on the UML *Property* metaclass.

2.3 Performance Modeling and Prediction

Performance prediction of software and hardware is the focus of various research approaches for quite a while now.

"Performance evaluation is concerned with the description, analysis and optimisation of the dynamic behaviour of computer and communication systems. This involves the investigation of the flow of data, and control information, within and between components of a system. The aim is to understand the behaviour of the system and identify the aspects of the system which are sensitive from a performance point of view." [Hil05]

The primary goal is to predict certain quality of service attributes of the system under investigation like network throughput, task execution behavior, or CPU utilization. This will help developers to identify problems or bottlenecks in the early stages of the development before costly changes must be made [SHK98].

The basis for such prediction approaches are low-level formal models such as Markov chains, queuing networks, stochastic Petri nets, and stochastic process algebra. Also the target domain varies from approach to approach, ranging from the prediction of cache misses in a CPU architecture up to network traffic in IP networks. Each approach has a different focus and underlying technique to provide the necessary evaluation data. They are spanning from embedded [Hap05, Wan06], non-embedded [WS02, BdMIS04], distributed [FCF+13, TP09, CHL+03, LWF08, FCF+13], with fieldbus [LF07, LF12, COH07, MDFF06b, HCÅ03] to just standalone PLCs [FH12, FHMB13]. In [Per06] an evaluation and comparison of performance analysis methods for distributed embedded systems is given. [BdMIS04] also gives an overview of several model-based prediction approaches for the development of software.

Figure 2.15 sketches an abstract performance prediction process. It is based on [Bec08], but replaces the software model with a model of the system. The *System Model* is modeled with common modeling approaches like SysML, UML, or CONSENS. Usually multiple disciplines and their developer are involved in the creation or refinement of this model. Performance specific information is typically not a part of this *System Model*. Therefore is it necessary to either create an additional or add performance specific annotations in the existing model. These annotations cover, among others, resources, resource demands, input parameters, and workload definitions. A popular example for such an annotation profile is MARTE (see section 2.3.1).

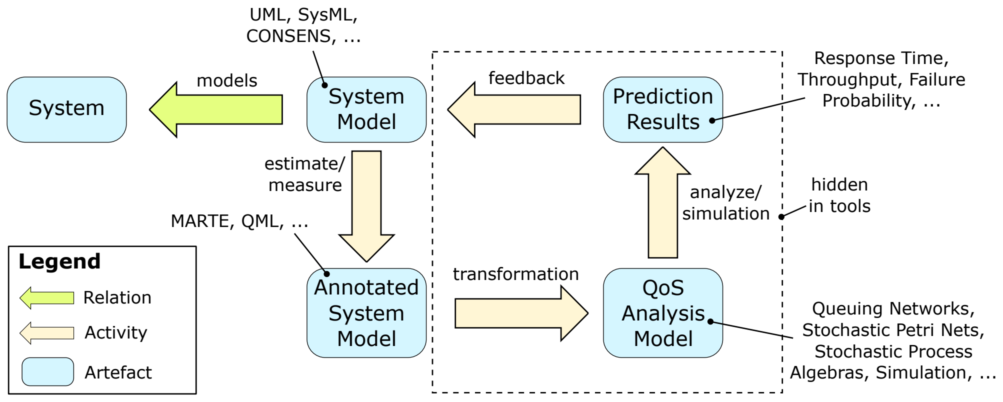


Figure 2.15: Model-based Performance Prediction Process (modified from [Bec08])

In case all necessary performance specific information has been provided, transformations can be executed that will transform the Annotated System Model into one or more *QoS Analysis Models*. These models can be formalized by Queuing Networks, Stochastic Petri Nets, Stochastic Process Algebras or specific Simulations. According tools use these transformed models as input to predict certain QoS attributes like response times or throughput. These results can be used by the developers to improve their original *System Model* up to the point where all requirements are met and the system can finally be realized.

To conduct a performance evaluation, usually three techniques are used: **Formal analysis** with analytical modeling, **simulations**, and measurements based on **performance prototypes**. Each technique has pros and cons, and the decision which one to use should be (but not not limited to) based on the life cycle stage the system is in [Jai90]. In the following, a brief introduction to three formal Queuing Networks (QN), Stochastic Process Algebras, and Stochastic Petri-nets is given. Afterward, the simulation and performance prototype are discussed.

Queuing Networks, short QN, are a powerful tool for system performance evaluation and prediction. Queues and their service centers represent processing resources which process job queues for a specific service. Queuing system models represent the system as a unique resource, whereas queuing networks represent the system as a set of interacting service centers. The latter are used to model system structure and to represent traffic flow among resources. This can be used to model jobs traveling through a network of queuing networks using probabilistic routes. The result of an analysis provides for example the average response time of the overall system, waiting times for individual queues, and utilization of the services.

Figure 2.16 provided by [Bec08] shows an exemplary queuing network that provides service to a set of customers. A QN is open, if customers can arrive at the system and leave it again. A constant number of customers as shown in the figure is called closed. Jobs queue at the CPU until they are processed. Afterward, they either use a hard-drive with a probability of 30% or a network resource with a probability of 70%. Due to the fact that this system is closed, all jobs return to the CPU after a short delay, indicated by the clock. Queuing Networks are the basis for a increasing number of performance prediction methods as surveyed by Balsamo et al. [BdMIS04].

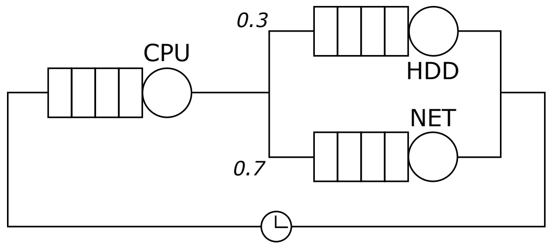


Figure 2.16: Example of a queuing network ([Bec08])

Petri Nets [Pet83] consist of a set of places and transitions. Transitions remove and add tokens on places whenever they fire. They cannot fire, if not in all places sufficient numbers of tokens are available. Stochastic Petri Nets[MBB+89] extend Petri Nets by random variables to represent the duration of activities, or delays until given events. In addition to that, probabilistic routing of tokens

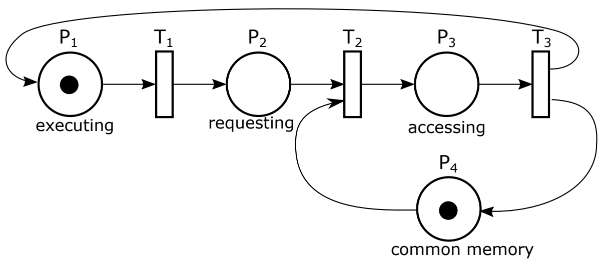


Figure 2.17: Simple Petri Net with four places (source [Hil09])

between the places can be specified. Figure 2.17 shows a simple example of a Petri Net that specifies four places (P_1 to P_4) and three transitions in between. The stochastic Petri Net represents a processor requesting and gaining access to the common memory. The example is taken from a teaching course of the University of Edinburgh created by Hillston [Hil09]. The processor executes for a given time and requests afterwards access to the memory. This memory could be used by other processors, consuming the available token and therefore inhibiting the firing of the transition T_3. Stochastic Petri Nets can be used to model and conduct a performance analysis [Mol82, MBC+94].

Stochastic Process algebras are abstract languages for specifying and designing concurrent systems. They are based on Milner's Calculus of Communicating Systems (CCS) [Mil89]. Their advantage is the ability to model the behavior of parallel processes formally. Therefore, routes through the model are not necessarily depended on probabilities, but behave to the semantics of the algebra. This further allows the analysis of other system properties like checking for deadlocks.

These performance prediction methods are used to model and analyze (solve) systems to evaluate certain quality of service attributes. Performance simulations are often based upon these methods. A simulation can incorporate more details from the real world at the expense of time it takes to come to results which are sufficiently precise. There exist a huge range of simulation tools used for different purposes.

TrueTime[CHL+03, HCÅ03, COH07] is a MATLAB [Mat16] and Simulink [Mat17] based simulator for real-time control systems. The framework allows the specification, programming, and simulation of programs, threads, real-time kernels, schedulers, network transmissions, and continuous plant dynamics. To check for the design of the network with regard to throughput, determinism, delays and jitter different network simulators like OMNeT++, OPNET, or NS2/NS3 [Ope17, VIN17, NS-17, VH08, Riv17] are available. Other approaches can be used to predict the response times of complex automation systems with control loops based on different network types and topologies [LF07, LF12, MDFF06b]. And lastly are several approaches tailored or focused on the analysis of program executions in general [BKR09, RBB+11] or with a focus on the automotive domain [TA17, AADG12]. In [MWD+05], a simulation based on Colored Petri Nets is used to predict end-to-end delays and compared to or extended with results form a formal analysis based on model checking. In addition, the UML-PSI tool by Marzolla Balsamo and Marzolla [BM03] derives an event-driven simulation from UML system models. Cortellessa et al. [CPR07] use annotated UML models and transform them into specifically designed simulation models. Palladio, the tool that has been selected for this thesis based on a set of requirements, is a simulation based approach. More details in section 6.2.1.

Prototyping is often used to check the applicability of a function or - in case of performance prediction - to get an idea how the system will behave under certain usage scenarios. The prototype implements a subset of functions, either fully or as mock ups, so that basic analysis can be conducted. The prototype has the advantage that multiple aspects of the systems design can be checked like user interaction, performance, or behavior. However, all functions that are not fully implemented might have a significant impact on the performance of the system. Prototypes can be used in the early phases of the development. They are usually more cost intensive than model based approaches due to additional programming and measurement tasks. This includes the creation of workload generators, test environments, and interfaces to external systems.

2.3.1 MARTE

The Modeling and Analysis of Real Time and Embedded systems (MARTE) [SG13, Obj06, HPV15] is the successor of the UML profile for Schedulability, Performance, and Time [Obj05]. The profile allows the annotation of existing UML models to further provide performance specific information with a focus on real-time and embedded devices. The profile contains four parts: a core framework defining the basic concepts required to support real-time and embedded domain, a package to support modeling of applications (hardware and software platform), quantitative analysis of UML2 models, textual language for value specification within UML2 model specially schedulability, and performance analysis.

MARTE supplements standard UML with the following basic capabilities

- Define and specify different types of quantitative and qualitative measures. This means that MARTE is capable of modeling different quality aspects of the (software) system, like network bandwidth, execution times of programs or arrival patterns between service calls.
- A time model to specify for example clocks, timepoints, and durations.
- A model to specify hardware resources like processors, memory, networks, input and output devices and many more.
- A model to specify software resources like threads, processes, or mutexes.
- The means to specify how software components are related to hardware components. This also covers the deployment of software onto hardware resources.

How MARTE can be used to model these different aspects of a complex system is briefly introduced with the following precise clock example, taken from the book “Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE” [SG13]. Figure 2.18 shows a class diagram with three classes. The *Timer* class has been extended by the *timerResource* stereotype. The additional properties this stereotype provides are used to specify that the *Timer* is executing or triggering a function of the *Displayer* every 100 us in a periodic way. The *Displayer* is a scheduable resource that uses a certain amount of processor resources when the *timeout()* function is called. In this case, the value specified is depended on a *CPU Speed* variable provided by an underlying hardware model and notated via MARTEs Value Specification Language (VSL). The last element shows a *hwDevice* namend *LEDDisplay* which is used to visualize the current time.

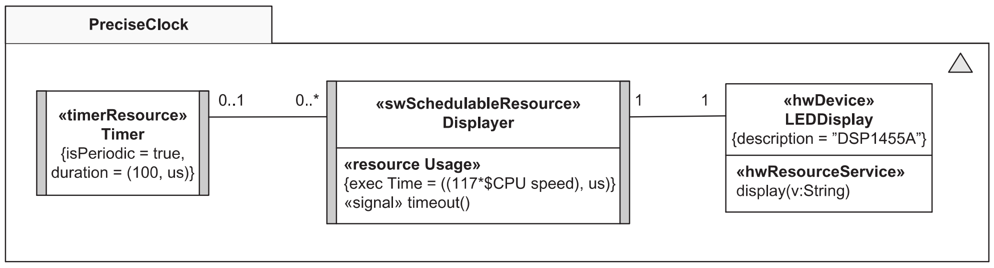


Figure 2.18: Precise clock class diagram (modified from [SG13])

Figure 2.19 shows the behavior of the system in form of a sequence diagram. It is also used to further define requirements the system must fulfill, by adding the *TimedConstraint* stereotype to the sequence diagram. It defines that the duration between time points t2 and t1 is required to be less than or equal to 100 ms. The Displayer creates a Timer, which will then call the timeout function every 100 ms back to the *Displayer*. The *TimedInstantObservation* stereotype is used to associate a clock with to time points (@t1 and @t2) to be able to validate the constraint.

An important foundation of MARTE is a rich set of basic definitions to formalize all aspects of the model, ranging from time to length as dimensions. For this, a special stereotype *Dimension* is used. A dimension in MARTE is a list of named units belonging to a given standard physical dimension. MARTE covers four dimensions: length (L), mass (M), time (T), and data (D). These can be combined to represent other more complex units, such as speed (length over time) or area (length squared), and so on. The standard set of dimensions and measurement units contain, among others, *LengthUnitKind*, *WeightUnitKind*, *DataSizeUnitKind*, *TimeUnitKind*, and *DataTxRateUnitKind*. An exerpt of these units are shown in Figure 2.20. The *TimeUnitKind* defines different kinds of time measurement units like *seconds (s)* and further more units that can refer to this base unit *ms* {*baseUnit=s, convFactor=0.001*}. Other measurements like the *tick* must be put into relation depending on the context (e.g. tick is every 1ms for a certain hardware device). The *convFactor* attribute allows to

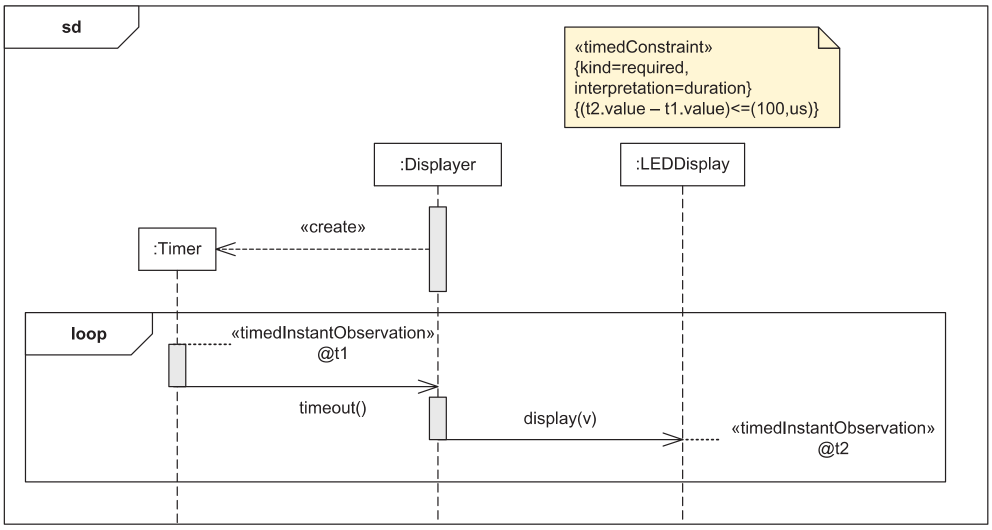


Figure 2.19: Precise clock sequence diagram ([SG13])

create more easily readable models and still set different values into relation. The three examples in the figure each represent a Time, Length, and Data dimension, denoted by the attribute *symbol*.

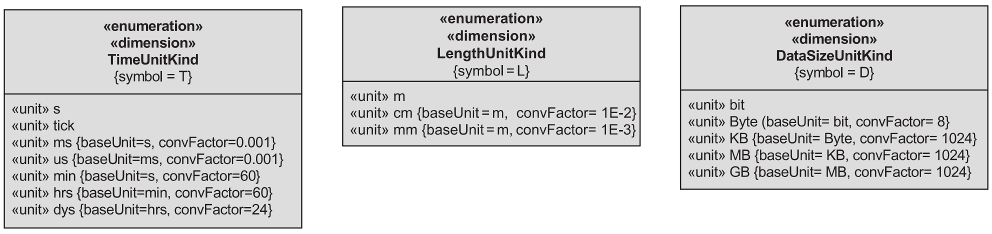

«enumeration»
«dimension»
TimeUnitKind
{symbol = T}
«unit» s
«unit» tick
«unit» ms {baseUnit=s, convFactor=0.001}
«unit» us {baseUnit=ms, convFactor=0.001}
«unit» min {baseUnit=s, convFactor=60}
«unit» hrs {baseUnit=min, convFactor=60}
«unit» dys {baseUnit=hrs, convFactor=24}
«enumeration»
«dimension»
LengthUnitKind
{symbol = L}
«unit» m
«unit» cm {baseUnit = m, convFactor= 1E-2}
«unit» mm {baseUnit = m, convFactor= 1E-3}
«enumeration»
«dimension»
DataSizeUnitKind
{symbol = D}
«unit» bit
«unit» Byte (baseUnit= bit, convFactor= 8}
«unit» KB {baseUnit= Byte, convFactor= 1024}
«unit» MB {baseUnit= KB, convFactor= 1024}
«unit» GB {baseUnit= MB, convFactor= 1024}

Figure 2.20: Exemplary set of dimensions and measurement units

To model various aspects of a real-time system, MARTE provides a set of non-functional property types (nfp types) and arrival patterns. These nfp types can be used to set data transfer rates (*NFP_DataTxRate*), Frequency (*NFP_Frequency*), or power (*NFP_Power*). To model the timing behavior of a system and its environment *ArrivalPatterns* can be used.

Figure 2.21 shows an exert of these patterns to model for example the frequency in which a user triggers functions or how IP packages arrive at the network interface of an embedded system. They provide means for modeling probabilistically-distributed workloads. MARTE provides a set of commonly used patterns as shown in the figure. A stimuli defined with a *PeriodicPattern* will be triggered in set intervals, respectively periods. The *PeriodicPattern* allows to set further properties like the jitter, which will bring in a variance from this fixed period length.

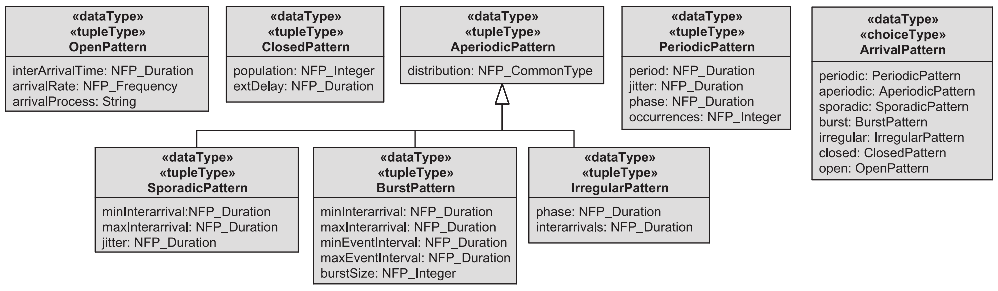


Figure 2.21: ArrivalPatterns to model probabilistically-distributed workloads

2.3.2 Palladio

The Palladio Component Model (further referenced just as Palladio) [RBB+11, Rec08, BKR09] *"is an architecture description language supporting design time performance evaluations of component-based software systems"* [Hap08]. Palladio provides a rich set of approaches to conduct performance analysis. Various transformations can be used to map the PCM to for example stochastic regular expression [FBH05], Layered Queueing Networks [Fra99], or event-based simulation frameworks [BKR07].

Core of the Palladio Component Model consists of five partial models, each targeted for a certain role in a component-based software engineering process. *Component developers* specify and implement components in a Repository model. *Software architects* compose the different components in a *System* model. The *System deployer* creates an *ResourceEnvironment model* containing platforms and networks and allocates/deploys the composed system onto the environment with the *Allocation model*. Finally the *Business domain experts* model the behavior of the users or external service that access the system with a *Usage model*. The five models are further detailed in this section.

These models are the basis for a rich set of transformations into further analysis tools and approaches. Focus of this thesis is **SimuCom**, a core part of Palladio. SimuCom performs a process and event-based simulation of the modeled system and allows to take measurements for QoS metrics like CPU utilization or response times. Simucom also provides fine grained data of the simulation, allowing the performance analyst to conduct in-depth investigations of the system under development. Via extension mechanisms, for example, custom operating system schedulers can be created and added to the Simcom simulation.

The **PCM Solver** is also integrated into the base package of Palladio and provides two sub-solver. The first is based on stochastic regular expressions (SRE) and provides a fast calculation of distribution function for one user. The second sub-solver is another analytical approach based on Layered Queueing Networks that allows a fast numeric approximation of performance metrics for multiple users and concurrent processes.

SimuLizar [BLB13, BBM13] is another Palladio plug-in for analyzing self-adaptive systems such as cloud computing systems. SimuLizar does not generate specific models but interprets the PCM input to provide measurements of for example response times or utilization. By interpreting the models, Simulizar can cope with changes during the simulation, hence the focus on self-adapting systems.

ProtoCom transforms PCM models into runnable Java code as a performance prototype that can be executed. During the execution, a set of sensors capture metrics of the system, allowing to conduct a performance analysis. These can be used for an early assessment of the modeled software system within a real environment.

Palladio Component Model

The models that are part of the Palladio Component Model are shown in Figure 2.22. On the left side, the five models and their relations between them are depicted. On the right side, the different transformations are sketched that generate the input models for each analysis approach. Instead of generating, Simulizar interprets existing Palladio models to perform its analysis. Other targets are input models for the Layered Queueing Networks (LQN) solver or a Stochastic Regular Expressions (SRE) notation for the SRE solver. Both are part of the standard Palladio PCM solvers [KG08]. In the following subsections, each model is briefly described.

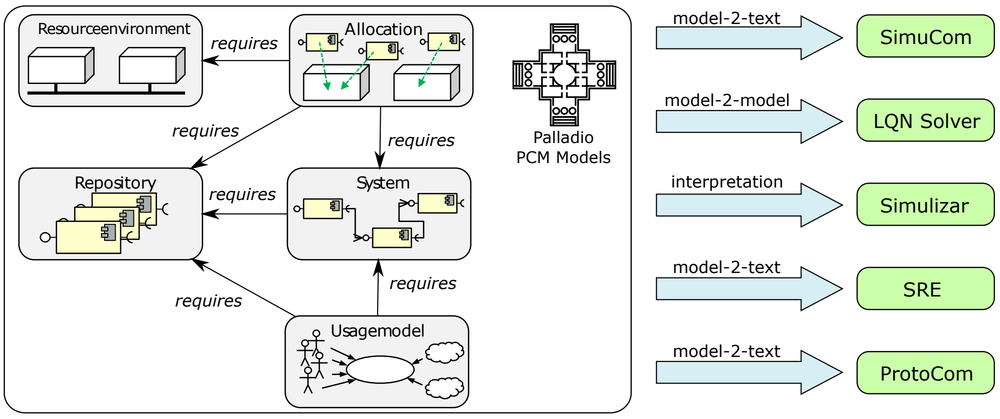


Figure 2.22: Palladio-Models used by different analysis tools

Repository Model

In the Repository Model, the developers specifies and implements components that are used in the *System model* to compose complex software. The major elements of this model are Interfaces, Components, and the Resource Demanding Service Effect Specifications. Operations and Parameters called between components must be specified by *Interfaces*. *Components* can provide or require one or more interfaces which serve as a contract between them.

Components can be categorized into atomic *BasicComponents* or *Composed-Components* which are created by composing Basic- or other ComposedComponents. Components must implement their interfaces by providing an abstract behavioral specification called Resource-Demanding Service-Effect-Specification (RD-SEFF). This is used to specify how the components use hardware/software resources during their operation and which other operations are called. *Internal actions* are used to specify the *Resource Demand* of a components operation. An abstract demand of a resource is modeled like CPU units needed or bytes read or written to a hard disk. These resources must be specified in the Resource Environment model, which will be detailed later. *External actions* are used to trigger operations on other Components. Other available actions allow the specification of complex control flows like *Loop*, *Fork*, *Acquire*, *Release* or *Branch* actions. Palladio provides an EBNF to further detail the resource usages and parametric dependencies. The following Probability Mass Function

$$data.BYTESIZE = IntPMF[(1000; 0.8)(2000; 0.2)]$$

sets the BYTESIZE property of the data variable to 1000 bytes in 80% of the executions and to 2000 for the remaining 20%.

System Model
The *System Model* is used to instantiate and assemble the different components. These instances, called assembly contexts, are connected via *SystemAssembly-Connectors*. The System itself provides interfaces that externals systems or users can access. All calls are delegated to the appropriate assembly contexts.

Resource Environment Model
This model is used to specify the hardware and environment of the overall system. It consists of *Resource Containers* which are connected via network links. Containers consists of multiple active and passive resources that can be referenced and used by the Components and their RD-SEFFS. A typical resource for a PLC is the CPU, which needs to specify a processing rate of any number of resource units per time unit (e.g. 64.000 resource units per millisecond). An active resource can also be configured with different schedulers/scheduling strategies, depending on the operating system and its settings.

Allocation Model
After the overall system has been composed in the *System model* and the underlying *Resource Environment* is set up, the allocation model is used. This model contains deployment information how each component of the system is assigned to a corresponding hardware resource/execution environment.

Usage Model
To model interactions with users or other external systems, the Usage model is used. It contains one or multiple *UsageScenarios* in which the call of one or more operations of the provided system interfaces is specified. Different workload profiles can be used to model the pattern and arrival times of these calls. Like RD-SEFF can UsageScenarios contain complex control flows with various loops and branches.

CHAPTER 3

Running Example

For this thesis, a medium sized production system provided by ELHA[1] to mill the body of Turbochargers is used as an ongoing example. To protect the intellectual property of ELHA, the provided System Engineering models have been modified to obfuscate the real system structure and its components. Figure 3.1 shows a photo of a similar system with two Mills. The input and output workpieces are transported by robots in-between. Several other components like stations to measure, buffer, or to engrave the workpieces are also part of this production system. Most of the stations can be reused. Therefore, a module oriented approach has been used by ELHA. A module can contain none or more stations and holds all structural and behavioral information. A module can be composed of other modules. Modules are controlled by one or many PLCs. The PLCs execute the Programs and Function Blocks to control, for example, the individual stations and the transport of the workpieces.

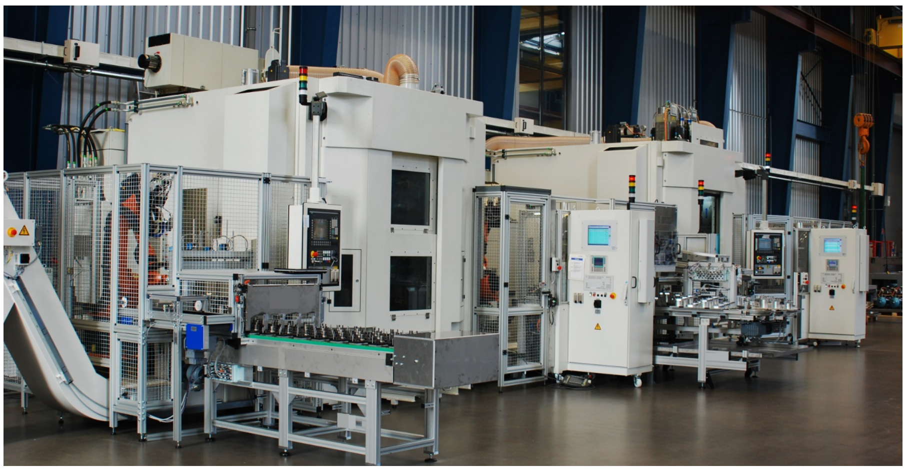


Figure 3.1: Photo of two milling centers with automation hardware for workpiece transport (source ELHA)

[1]ELHA Maschinenbau – http://www.elha.de/

3.1 Overview

The overall composition of the system is shown in the active structure Figure 3.2 – please note that the model has been heavily simplified to give just an overview of some of the different parts. Also, all flows (information, material, and electrical energy) have been hidden to improve the visibility. The figure shows the different stations and indicates the three main modules of this automation system. A detailed excerpt of the right part (Unit2) is shown in Figure 3.3. The active structure is one of multiple models to specify the CONSENS principle solution. It is used to give a hierarchical overview of the system structure and the relations between the different elements. *Environment elements* (yellow) indicate an element that is not part of the system under development, but which influences it. In case of the turbocharger production system, the typical environment elements are chippings from the mill, operators working with the production system, external remote diagnostic systems accessing the internal data, or sources for water, air, oil, and electrical energy. The *system element* (blue) is used to model the structure of the system under development. Each system element can be further composed of more elements on a lower hierarchy level, detailing the model element. The relations between elements can be specified in more detail, for which the active structure model uses different types: the electrical flow to indicate that two elements are connected with a power connector, the material flow to show that some kind of material is exchanged between elements, like workpieces or cooling water, and the information flow which is used to model communication between elements.

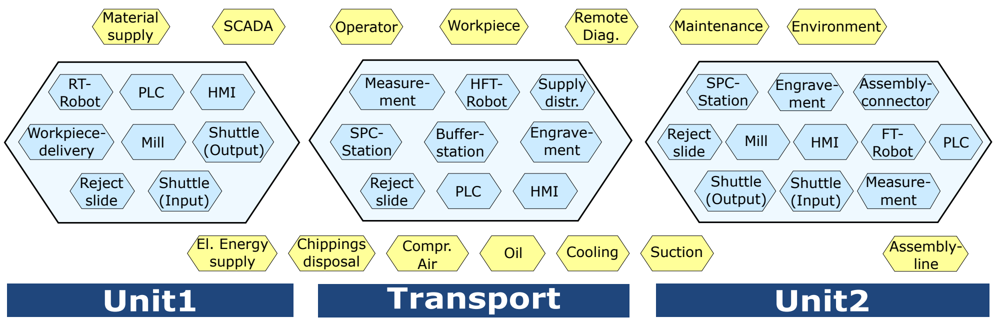


Figure 3.2: Overview of the ELHA production system elements

The Figure 3.2 shows two hierarchy levels of the production system and selected environment elements. At the top level, three system elements can be identified. First is the *Unit1* in which workpieces gets loaded and checked. The *Unit1* also includes the first *mill* and two shuttles used to deliver and take out workpieces of the mill. The shuttles are loaded and unloaded by a robot arm. The second top-level element is the *Transport* unit. When the first processing step of the workpieces is finished, the *shuttles* transport the workpiece out of the mill. A *robot* called HFT-Robot (Halb-Fertig-Teil) takes the workpiece from the shuttle and puts it on an *engraving station*. There, a serial number and other information is engraved into the piece before the same robot delivers it to the *measurement station*. If the setpoints deviate from the actual values, the robot takes the workpiece and puts it on a *Reject slide* where the operator will manually remove it from the production process. If the next mill is ready to process the next workpiece, the robot will place it onto the shuttle of *Unit2*. In case the mill is not ready, the workpiece will be delivered to a *Buffer station*. *Unit2* is a mixture of *Unit1* and the *Transport* unit. It includes a mill and shuttles, a second engraving and measurement station to check the quality of the second milling step and several other modules that can be found in the previous two parts. If the workpiece is within the set production tolerances, another robot of *Unit2* will put the workpiece on a special belt where it will be moved to the next step in the production line, which is not part of the ELHA production system.

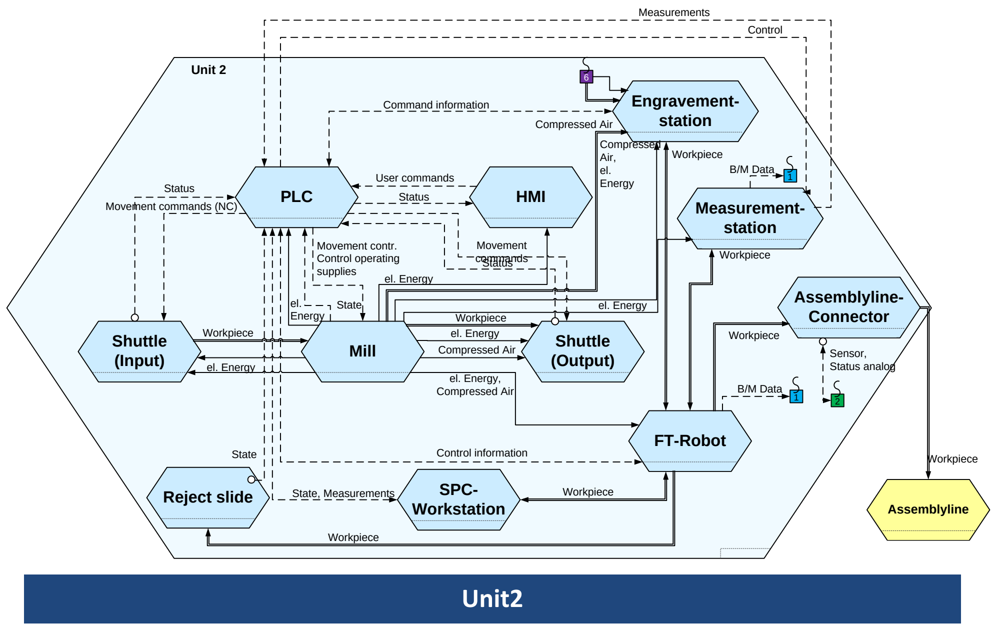


Figure 3.3: A simplified CONSENS active structure of Unit2

Figure 3.3 shows a more detailed view of *Unit2*. In this figure, some of the material, energy, and information flows are left out to reduce the complexity and improve visibility. The following list details some elements for a better understanding of the example system.

- **Mill:** The mill is the center of the production system. The workpieces coming from the foundry are rough and need to be drilled, ground, milled, and polished. The mill is a closed system with input and output gates on the left and right side. The shuttles are used to transport the workpiece through the gates into the mill and out again. Inside the mill, multiple tools can be used to process the workpiece.
- **Workpiece delivery:** This part is used to order the workpieces and put them on predefined places where a robot can pick them up. Several sensors check for availability and position of the workpieces. A conveyor

system is used to push the delivered workpieces to the front.

- **Shuttle:** The shuttle is a component that is used multiple times. It transports the workpiece in and out of the mill. For the transport, the workpieces must be locked. To do this, several sensors and actuators are used. Depending on the complexity of the workpiece, a phased or staged locking process in close cooperation and/or synchronization with the robot is necessary.
- **Robot:** For the turbocharger production line, three robots are used. The *RT-Robot* (Rohteil-Robot) has the simple task of taking the workpiece from the workpiece delivery and to place it on the shuttle. The second *HFT-Robot* (Halb-Fertig-Teil) has tasks that needs to be executed in a specific order and which are highly time critical to ensure the throughput of the production line. Some of these tasks are to take the workpiece from the shuttle, deliver to the engraving station, to the measurement station, to the reject slide, to the buffer station, to the SPC-Workstation and to the shuttle of the *Unit2* mill. The last robot is the *FT-Robot* (Fertig-Teil) which has to perform the same tasks as the second robot; but instead of delivering the workpiece to another shuttle it has to deploy it on a conveyor belt.
- **Engraving-Station:** This station is used to engrave the workpiece. A serial number and additional information are engraved with a laser or a drill. Like the shuttle, it is necessary to lock the workpieces for processing into place. This locking might also be multi-staged and a synchronization with the robot delivering the workpiece could be needed.
- **Measurement-Station:** To check whether the production tolerances are adhered after the milling process, the measurement station can automatically measure predefined distances or faces. The measurement data is stored in the station until it is collected by the SCADA system.
- **Buffer-Station:** If the *Unit2* mill is under stress and cannot process workpiece as fast the *Unit1* mill, all accumulated pieces can be stored in a buffer station. The amount of pieces that can be stored depends on the type of buffer station.
- **Reject slide:** The slide is just used to remove damaged workpieces or the ones which were not in the tolerance range after the measurement. The operator of the system needs to manually take the pieces from the slide. No actuators are used, just a single sensor at the top to check whether the slide is still free or full.
- **HMI:** One of the most important ways to influence and control the system is via the human machine interface (HMI). The current status of all the components and the overall system can be queried here. Often provides the HMI means to switch between manual and automatic mode. Manual mode is usually used to find problems in the production process because each individual component can be accessed and modified step by step.
- **PLC:** Heart of the automation system is usually one or more programmable logic controller (PLC). The PLC gathers data from the sensors, calculates the next steps to perform and sends the command/data to the actuators. Complex automation systems consist of a network of PLCs ex-

changing data. Details on PLCs and programming of automation systems can be found in Chapter 2.
- **SCADA:** The Supervisory Control and Data Acquisition (SCADA) system used for remote monitoring and control of the turbocharger production. In addition, it monitors industrial processes for display or for recording functions. The SCADA system is an environment element and therefore not shown the Figure 3.3.
- **Remote-Diagnostics:** Like the SCADA system are remote diagnostic systems used to access the current state of the overall system as well as individual components. It allows the maintainer to check for problems without the need to physically access the automation system. Like the SCADA system is the Remote-Diagnostics an environment element that accesses several subelements of the Unit2 and not shown in Figure 3.3.

3.2 CONSENS Models

In the following sections, the different parts and a detailed structure of the Turbocharger production system is shown. To protect the intellectual property of ELHA, the system structure and its components are obfuscated. The diagrams show excerpts of the full CONSENS model. Most labels of connectors are removed to improve the overall visibility. This helps to focus on the important parts of the Turbocharger model. The original software of the production system contains more than 370 Function Blocks. They are hierarchically structured and contained in round about 35 entry level Function Blocks. For this thesis, this huge amount of software components has been reduced to only a few, selected elements. This will further reduce the complexity of the model and allows the reader to focus on the techniques and concepts. A full list of all used SystemElements, EnvironmentElements, Tasks, Function Blocks, and Programs can be reviewed in the Appendix 3. The models shown in the following subsections have been created with the SysML4CONSENS profile [KDHM13] as introduced in Chapter 2. The profile extends SysML by a selection of CONSENS specific stereotypes. The models are created with the Papyrus Modeling environment [Ecl17c] (as part of the Eclipse IDE [Ecl17a]).

3.2.1 Environment

The environment model of the turbocharger production system is depicted in Figure 3.4. The SystemElement *TurboChargerMill* (blue) in the center interacts via flows with the various EnvironmentElements (yellow) in its surrounding. EnvironmentElements indicate an element that is not part of the system under development but which influences it. The environment model (as well as the following active structure models) provide three basic kinds of flows for specifying an interaction between elements: the electrical flow to indicate that two elements are connected with a power connector, the material flow to show that some kind of material is exchanged between elements, like for example workpieces, chippings or cooling water, and the information flow which is used to model communication between elements of the active structure.

In case of the turbocharger production system, typical environment elements are chippings from the mill, operators working with the production system, external remote diagnostic systems accessing the internal data or sources for water, air, oil or electrical energy. Other elements like the *CompressedAirSupply* or the *OilSupply* use material flows to indicate that some kind of material is exchanged between the elements. And the *ElectricalEnergySupply* provides energy to the SystemElement with the *ele.Energy* flow.

The *SCADA* and *RemoteDiagnostics* environment elements are two elements often used as examples throughout this thesis. Both elements aquire data from the system via information flows. The *SCADA-Data* transfers data in form of variables from the PLC to the SCADA-System and back again. The *Diagnostics* information flow is used to collect log files from the PLC so that they can be analyzed and stored.

The kind of flow is depending on the developers perspective. A fieldbus is used to exchange sensor and actuator data between different elements. However, it is also possible to use the fieldbus as an energy supply for the IOs. In this case a developer might choose to model an energy flow instead of an information flow. The selection of a flow kind depends on the intended use and should be fixed throughout the model by modeling guidelines.

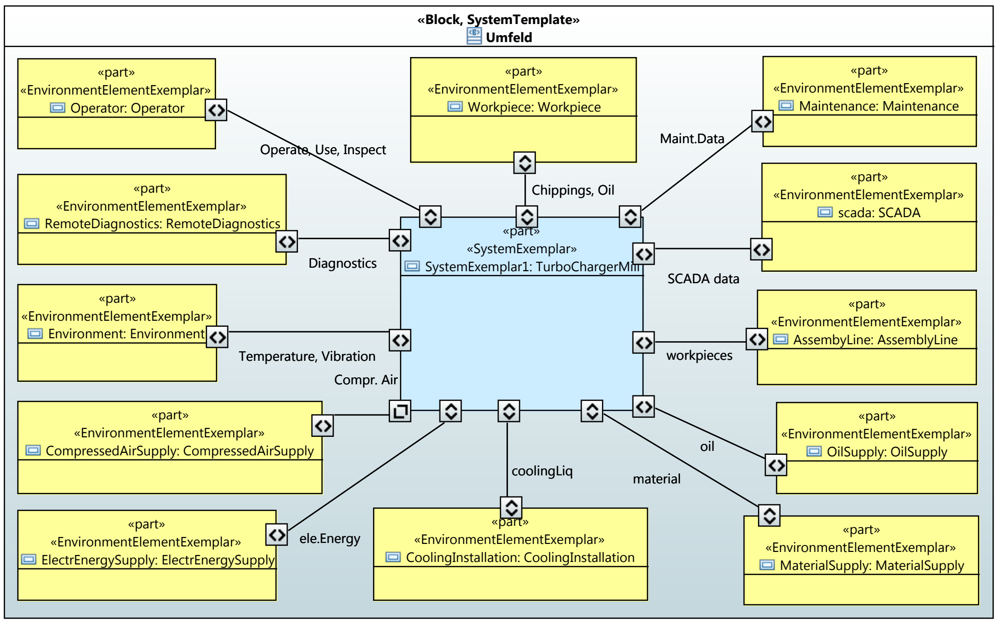

«Block, SystemTemplate»
Umfeld
«part» «EnvironmentElementExemplar» Operator: Operator
«part» «EnvironmentElementExemplar» Workpiece: Workpiece
«part» «EnvironmentElementExemplar» Maintenance: Maintenance
«part» «EnvironmentElementExemplar» RemoteDiagnostics: RemoteDiagnostics
«part» «EnvironmentElementExemplar» scada: SCADA
«part» «SystemExemplar» SystemExemplar1: TurboChargerMill
«part» «EnvironmentElementExemplar» Environment: Environment
«part» «EnvironmentElementExemplar» AssembyLine: AssemblyLine
«part» «EnvironmentElementExemplar» CompressedAirSupply: CompressedAirSupply
«part» «EnvironmentElementExemplar» OilSupply: OilSupply
«part» «EnvironmentElementExemplar» ElectrEnergySupply: ElectrEnergySupply
«part» «EnvironmentElementExemplar» CoolingInstallation: CoolingInstallation
«part» «EnvironmentElementExemplar» MaterialSupply: MaterialSupply
Operate, Use, Inspect
Chippings, Oil
Maint.Data
Diagnostics
SCADA data
Temperature, Vibration
workpieces
Compr. Air
oil
coolingLiq
material
ele.Energy

Figure 3.4: Environment model for the Turbocharger example (simplified)

3.2.2 Unit2 of the turbocharger production system

The *TurboChargerMill* is a top level SystemElement that is further decomposed into *Unit1*, *Transportation*, and *Unit2*. This hierarchy level is skipped to directly detail the containing elements of *Unit2* in Figure 3.5. Similar to the environment model diagram, this figure shows only selected parts of the model to improve the overall visibility.

The *Unit2* consists of thirteen SystemElements modeled as parts. These parts, as well as their according types specified as SysML Blocks, are annotated by SysML4CONSENS stereotypes. In the center of Figure 3.5, the PLC – focus of this thesis – is shown. The compartment of the PLC includes parts of the PLC: a *WebServer*, an *OPC-UAServer*, and an *FTPServer*. The parts are also typed by their according blocks. The PLC contains also contains the *MainTask* and the *MillTask*. These elements will be explained in more details in Chapter 4 and Chapter 6. The figure visualizes two outer ports of *Unit2* that are used to delegate information flows to the different subparts of the PLC. These flows are the Diagnostics and SCADA-Data flows that have been introduced in the environment model. Ports from the EnvironmentElements, *OPC-UAServer* and *FTPServer* are typed by corresponding information flows. The *HMI* uses data provided by the PLC to visualize the current state of the *Unit2*. This access is realized via a Webserver which transfers the data to the HMI via the *HMI Data* information flow.

There are several ways to model a data exchange between elements, depending on the level of detail that needs to be achieved for the current modeling purposes. The first one is to model each data exchange with an information flow connecting the two communicating partners directly. An example for this approach is the data exchange between the SCADA-System and the OPC-UA-Server. The second alternative is an explicit modeling of the communication system or media in use. In Figure 3.5, a SystemElement *Interbus* has been created to represent the underlying (field)bus for exchanging data. Each fieldbus specific Informationflow is routed over this element. Especially in automation systems, in which fieldbuses are an important part of the system structure, an explicit modeling of buses is advised. A complex production system may consist of several, separated buses for different communication purposes like fast motion control and data exchange over the Internet. Despite the fact that the element and the flows have already been named “interbus”, the influence model specific information to type the bus are not yet added to the model (see Chapter 5).

3.2.3 AssemblyConnector in detail

The Figure 3.6 further refines the *AssemblyConnector* SystemElement. The AssemblyConnector is used to put the finished workpieces onto a carrier that transports them to the next production system. Each carrier can transport three workpieces which must be fixed during transport. For this task, a left and

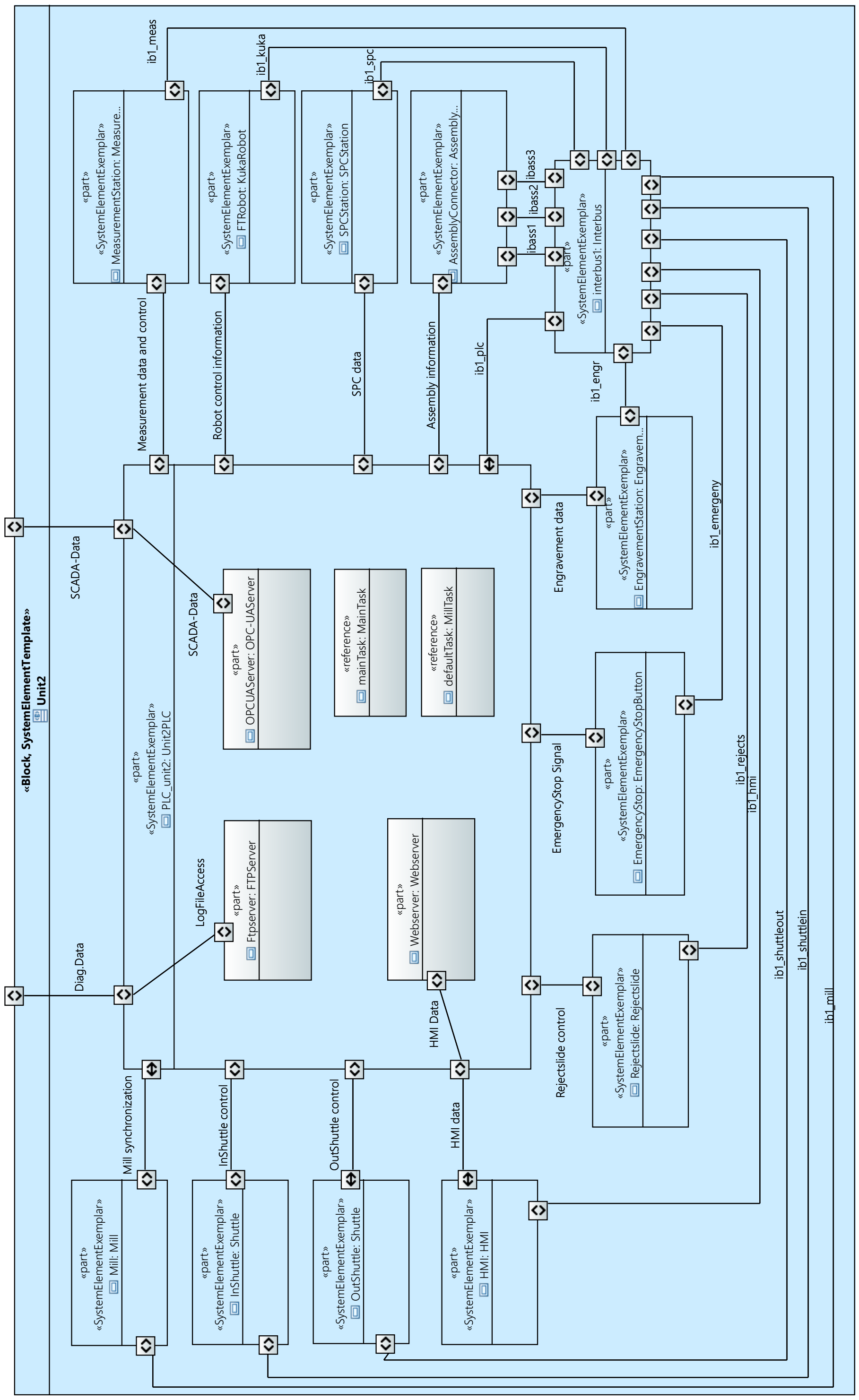

«Block, SystemElementTemplate»
Unit2
«part» «SystemElementExemplar» MeasurementStation: Measure...
«part» «SystemElementExemplar» FTRobot: KukaRobot
«part» «SystemElementExemplar» SPCStation: SPCStation
«part» «SystemElementExemplar» AssemblyConnector: Assembly...
«part» «SystemElementExemplar» interbus1: Interbus
«part» «SystemElementExemplar» PLC_unit2: Unit2PLC
«part» OPCUAServer: OPC-UAServer
«reference» mainTask: MainTask
«reference» defaultTask: MillTask
«part» Ftpserver: FTPServer
«part» Webserver: Webserver
«part» «SystemElementExemplar» EngravementStation: Engravem...
«part» «SystemElementExemplar» EmergencyStop: EmergencyStopButton
«part» «SystemElementExemplar» Rejectslide: Rejectslide
«part» «SystemElementExemplar» Mill: Mill
«part» «SystemElementExemplar» InShuttle: Shuttle
«part» «SystemElementExemplar» OutShuttle: Shuttle
«part» «SystemElementExemplar» HMI: HMI
ib1_meas
ib1_kuka
ib1_spc
ibass1
ibass2
ibass3
ib1_plc
ib1_engr
ib1_emergeny
ib1_rejects
ib1_hmi
ib1_shuttleout
ib1_shuttlein
ib1_mill
Measurement data and control
Robot control information
SPC data
Assembly information
SCADA-Data
Diag.Data
LogFileAccess
Engravement data
EmergencyStop Signal
Rejectslide control
HMI Data
HMI data
Mill synchronization
InShuttle control
OutShuttle control

Figure 3.5: Excerpt of the *Unit2* System element of the Turbocharger example

right *Fixing* for each lane is used. Also a *LaneSensor* checks for empty/full lanes on each carrier. An underlying conveyor belt runs continuously. Therefore, a stopper is used to stop carriers in a certain position. The raiser slightly raises the workpiece carrier so that no vibration from the conveyor belt is transfered to the workpiece carrier during the placement of the workpieces by the robot. The carrier is only send to the next production system if no queue exists in front of it. Therefore, an EmptyQueue sensor is used to send a 'lane free' to release the stopper. The whole AssemblyConnector contains eight actuators, four sensors and three buscouplers to send data from and to the PLC. The data is exchanged via information flows that are typed by the ports. The labels of the ports and connections are omitted in the figure.

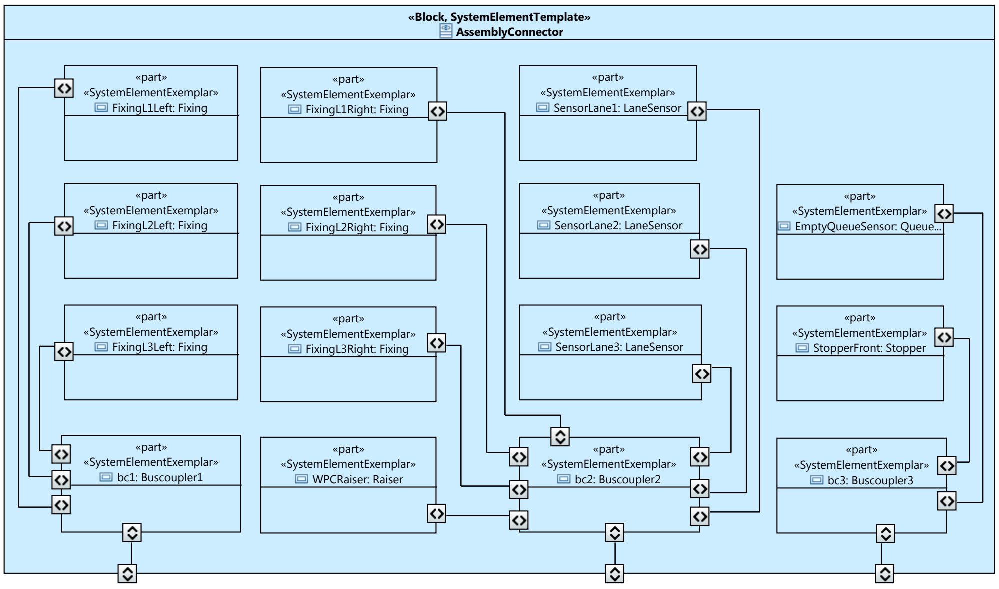

«Block, SystemElementTemplate»
AssemblyConnector
«part» «SystemElementExemplar» FixingL1Left: Fixing
«part» «SystemElementExemplar» FixingL1Right: Fixing
«part» «SystemElementExemplar» SensorLane1: LaneSensor
«part» «SystemElementExemplar» FixingL2Left: Fixing
«part» «SystemElementExemplar» FixingL2Right: Fixing
«part» «SystemElementExemplar» SensorLane2: LaneSensor
«part» «SystemElementExemplar» EmptyQueueSensor: Queue...
«part» «SystemElementExemplar» FixingL3Left: Fixing
«part» «SystemElementExemplar» FixingL3Right: Fixing
«part» «SystemElementExemplar» SensorLane3: LaneSensor
«part» «SystemElementExemplar» StopperFront: Stopper
«part» «SystemElementExemplar» bc1: Buscoupler1
«part» «SystemElementExemplar» WPCRaiser: Raiser
«part» «SystemElementExemplar» bc2: Buscoupler2
«part» «SystemElementExemplar» bc3: Buscoupler3

Figure 3.6: The *AssemblyConnector* and various sensors and actuators

CHAPTER 4

Influence Factors

In this chapter, the different influence factors that impact the overall utilization or throughput of an automation system are identified and suitable representations are specified. Several hardware, software, and environment related influence factors exist that need to be considered when developing a new or updating an existing system. The factors and their parameters will be described in detail in the following subsections. Depending on the influence of the factor, the ease to model it, and if the necessary information is already available in the early development phases, the factors are either neglected, simplified, or considered in full detail.

The list of influence factors has been gathered by analyzing common factors from existing performance prediction approaches, by exhibiting the firmware of a well know PLC vendor (Phoenix Contact), and by analyzing exemplary System Engineering models of automation systems as shown in Figure 4.1. One of these exemplary systems is the ELHA turbocharger milling system which is introduced in Chapter 3. However, the major part of the identified influence factors has been obtained during a two-year long project with Phoenix Contact. The goal of this project was to develop a tool to support a sales person to chose a fitting PLC performance class for a given customer automation system. Most of the measurements and detailed information are under restricted confidentiality. Still, the experience gained in this project is used to support the decisions when identifying and rating the influence factors. Several presentations, posters, and publications (e.g. [FH12, FHMB13]) are results of this industry-driven project.

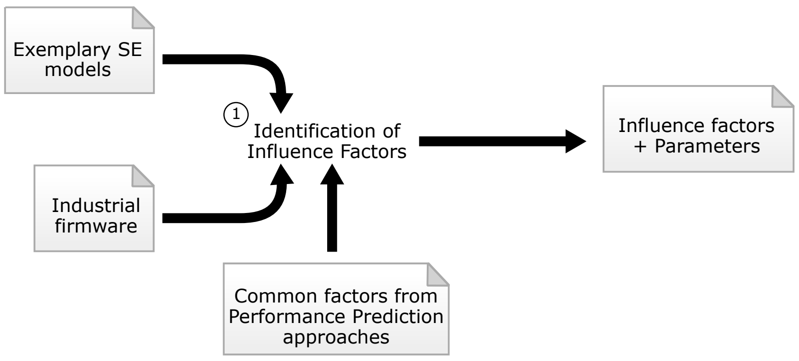

Exemplary SE models
1
Identification of Influence Factors
Influence factors + Parameters
Industrial firmware
Common factors from Performance Prediction approaches

Figure 4.1: Identification of influence factors and artifacts

Contribution C1:
Identification of Influence Factors
The contribution of this chapter is the identification of influence factors that impact quality of service (QoS) attributes of an automation system like the CPU utilization of a PLC. An extensive list will be gathered that details each influence factor. It will be discussed, whether information about the influence factors is available in the early development stages, which assumptions have to be made, its overall impact on the system, and which parameters need to be taken into account.

This chapter is structured as follows. First, a selection of related work is presented (see Section 4.1), providing a general overview of different performance prediction approaches and the domains they provide influence factors for. Afterward, the foundation for describing the influence factors is set by defining used parameters and notations, starting with Section 4.2. Focus of this chapter are the following influence factors: The application related influence factors are described in more detail in Section 4.3. These factors include - among others - IEC 61131-3 related programs, functions, and task deployment settings. They are usually independent of the PLC they are executed on. The second topic are hardware dependent (4.4) influence factors like architecture, operating system, firmware, and CPU. The input-output-systems (IO) has a huge impact on performance of a PLC and is discussed in the third part (Section 4.5). The fourth Section 4.6 deals with additional services of a PLC such as an FTPServer or an OPC-UA server. They could be categorized into the firmware (and thus associated with the section PLC), but are often used very application-specific. Since services have a non-negligible impact on the PLC performance, these factors have to be considered separately. The last section of this chapter (Section 4.7) gives an overview of all influence factors and their associated parameters. The results of this first identification are captured in a formal model in Chapter 5 and later on transferred into the UML profile presented in Chapter 6.

Summary:
The core influence factors which must be considered when conducting an analysis and/or developing an automation system are listed below. Most details are only important for a precise analysis of a PLC or side effects in a network or fieldbus. For example, even small changes in the compiler settings or task priorities can influence the execution behavior of a task and its programs. But these effects are usually hard to model or details are simply not available in the early stages of the development. Forcing developers to consider each and every little aspect will reduce the overall usability and lead to longer specification and analysis times [Jai90]. Also, by providing too much detail, the performance prediction will become harder, leading to wrong results. Starting with more abstract, high-level models to perform performance predictions and then detail these models to conduct in-depth analysis should be preferred [Jai90].

- **Program:** The execution of Program code directly influences the utilization of the PLC and the automations systems performance

- **Function Block:** Invoking a Function Block and its optional background load puts stress on the PLCs CPU.
- **Cyclic Task:** The CyclicTask is a periodic task with fixed intervals. The task is used to trigger the execution of a Program or Function Block.
- **Event Task:** This aperiodic task is hard to model due to the different sources for events which trigger the execution. An approximation with density functions is best to incorporate their induced utilization.
- **Idle Task:** The idle task is dependent on the execution time of associated Programs and Function Blocks. The interval of the task may jitter.
- **PLC:** The PLC as a container for several hardware related factors like CPU frequency, caching, architecture and multi core influences the execution of Programs and Function Blocks, services, and all other computational factors.
- **OS:** Operating system processes create a base utilization on the PLC. For embedded devices these processes are stripped down to a necessary core. The base utilization can be neglected, but the impact of this factor is low.
- **Firmware & Runtime:** Management processes and Runtime Environment influence the code execution and create a background utilization. Usually the performance oriented design puts low stress on the CPU and is therefore a low impact factor.
- **IPTraffic:** The major part of IPTraffic is covered in the IO and the service influence factors. However, a tight integration with an office network or attacks may (in rare cases) affect the automation system
- **IO:** Receiving, processing and sending of process data is are CPU intensive tasks. Their impact on the performance of a PLC are high. Several parameters need to be considered.
- **Services:** There are various types of services already available and in development for future PLC generations. They become more and more popular, hence their influence on the PLCs performance can not be neglected. When and to what degree the performance of the PLC is influenced is depending on the service and the vendor.

4.1 Related Work

To identify influence factors three primary sources are taken into account: Exemplary automation systems, an in-depth analysis of an industrial PLC firmware and various literature for performance modeling. This literature can be broadly categorized into general approaches (e.g. how to model factors and perform analysis) and domain specific approaches (e.g. to predict the energy consumption of a fieldbus). Each approach has a different focus and underlying techniques to provide the necessary evaluation data. They span from embedded [Hap05, Wan06], non-embedded [WS02], distributed [TP09, CHL+03, LWF08, FCF+13], with fieldbus [LF07, LF12, COH07, MDFF06b, HCÅ03] to just standalone PLCs [FH12, FHMB13]. In [Per06], an evaluation and comparison of performance analysis methods for distributed embedded systems are given. Another survey compares different model-based performance prediction approaches[BdMIS04]. The majority of these approaches only provide means to model and simulate a specific aspect or part of the automation system, but do not focus on the identification of factors. However, they usually provide examples which can be used as further input for the identification process.

This thesis focuses on the application domain of industrial control systems. The approaches listed provide different kinds of influence factors and parameters for their specific uses. They are used as a basis and give a first insight to some of these factors which must be considered in automation systems. Additional, in-depth related work is referenced when inspecting each influence factor and its parameters in detail in the following sections. The listed approaches are to a part also presented in the related work sections of other Chapters, but investigated under different aspects like development process, their formal model, or simulation approach/tool support.

“The art of computer systems performance analysis” [Jai90] by Rai Jain is a basis literature for conducting performance analysis. He presents general aspects on how to prepare, conduct, and interpret the results of an analysis. Different kinds of techniques are discussed and advantages and disadvantages for specific purposes are presented. The book does not introduce a common set of influence factors but explains how to identify, rate, and model them. Generally applicable examples help to identify automation specific influence factors as well as the metrics to capture them.

The Modeling and Analysis of Real-Time and Embedded systems (MARTE) [Obj06] is the successor of the UML profile for Schedulability, Performance, and Time (SPTP) [Obj05]. MARTE allows the annotation of existing UML models to further provide performance specific information with a focus on real-time and embedded devices. The profile allows the definition of elements like semaphores, concurrent tasks, or schedulers and scheduling policies. Software elements can be annotated with execution times, resource usages, and relations among each other. The pool of resources contains for example processors, memory, input and output devices, and networks. It is a general approach covering a broad range of embedded, real-time systems and their factors.

UML-RT [Sel98, KHCD17] is a UML profile designed to model real-time systems. It introduces *Capsules* and *Protocols* to model structure and behavior of an (embedded) system. Capsules are classes that own a behavior specified via state machines and which communicate via messages send through ports. These ports are typed via Protocols that explicitly define the kind and order of messages that can be send and received by the port. While the core of UML-RT is based on these elements, it also provides elements to model more complex systems like Service Access Point (SAP) or Service Provision Point (SPP). While the specification of timing constraints and durations is possible, detailed option to model arrival times or service access is not supported.

The simulation frameworks OMNeT++ [Ope17, VH08], NS2 [IH08, VIN17], and its predecessor NS3 [RH10] are focused on network analysis. These approaches are used to model the in-depth structure and behavior of TCP, routing, and multicast protocols over wired and wireless networks. Their modular approach allows the definition of further modules that can represent tasks and other properties of embedded devices. However, the influence factors and parameters they initially provide are network specific and cover delays, latencies, messages sizes, and more.

The Palladio Component Model (PCM) [BKR09, RBB^{+}11] is an architecture description language supporting performance evaluations of component-based software systems. Palladio provides mean to specify distributed processing resources connected via networks as well as passive resources. CPUs and hard disks are examples of processing resources, connections to a database or memory are passive resources. Palladio also offers the ability to model different scheduler for processing resources like an operating system. The OS can use scheduler to run the processes and assign them to different CPUs or cores. For this, basic scheduling algorithms and a framework to specify custom scheduler (see [Hap08, Hap04, Hap16]) can be used. Palladio further specifies a set of common units like file sizes. More details on Palladio are provided in the foundations Chapter 2 and in Chapter 6.

With the TrueTime library [CHL^{+}03, HCÅ03, COH07] for MATLAB [Mat16] and Simulink [Mat17], an automation domain targeted approach is presented. The framework allows the specification, programming, and simulation of programs, threads, real-time kernels, schedulers, network transmissions, as well as continuous plant dynamics. The library provides blocks which must be instantiated and parameterized. Such a block can abstract a complete fieldbus used to transport control commands and data or parts of an operating system. The influence factors available in TrueTime are to a large part congruent to the identified factors gathered during the analysis of the Phoenix Contact firmware.

In UML-Based Performance Modeling Framework for Component-Based Distributed Systems [Kah01] an approach to model performance relevant information in existing UML models. These models are then transformed into a textual notation that only holds performance relevant information. To conduct a performance analysis, the intermediate notation is again transformed into augmented queuing networks and solved. They show how to model distributed software systems considering three basic factors: CPU usage, hard disc access, and network traffic. These metrics are used for the definitions of functions and operations which are called during the systems runtime. The order of calls is specified via sequence diagrams, their workloads are modeled with collaboration diagrams. Their approach uses an abstract notation of general factors like the CPU, but allows to define complex ones by combining basic types.

A similar approach based on Modelica [Mod17b] is presented by Frey et al. [LWF08, FL09]. Their goal is to conduct open-loop response time analysis as well as closed-loop analysis of networked control systems. Furthermore, they developed a library for Modelica that allows the definition of distributed automation systems, but with a focus on the underlying fieldbus. Components provided by the library are for example Ethernet or CAN networks, CPU, memory, caches, or data structures. Their controller library allows the definition of schedulers including different scheduling strategies. By using the modular Modelica elements, even complex systems can be created by hierarchically composing components.

In [PMDB14], the focus is set on the model driven development of safety-critical systems. They define viewpoints and provide a method for the multi-view modeling of hardware platforms. By integrating their viewpoints and the support for hierarchical and variable horizontal composition of hardware platforms into the MechatronicUML [DPP^{+}16] modeling language, they enable developers to generate runnable code for the modeled targets. To do this, they provide a resource model that allows the specification and parameterization of embedded devices like micro controllers or control units. This model can be composed of atomic computing resources like processor, memory, flash or communication resources. Such resources can include their specific physical layer attributes like data rates and protocols. The hardware model in combination with software annotated by WCETs allows a subsequent performance analysis of the system under development.

Wanderler provides in his thesis [Wan06] another approach to conduct performance analysis based on a modular and interface-based design for embedded real-time systems. The approach is also based on MATLAB and provides a toolbox named Real-Time Calculus (RTC) Toolbox. RTC provides libraries to perform a modular, interface-based design and a subsequent performance analysis based on variability characterization curves (VCCs) [MZCW04]. To do so, the approach analyzes the flow of event streams over resources to derive performance characteristics of the modeled system. The thesis is focused on basic techniques to conduct a performance analysis and therefore presents only a few influence factors like CPUs, buses, and basic scheduling strategies that are used as examples in his work.

In [FCS12], Feljan, Carlson and Seceleanu propose a performance prediction approach for allocating tasks to multicore processors. They use Matlab to simulate a set of tasks allocated to CPUs and cores. Input for the simulation in an architectural specification of the system including tasks and connections. Each task has a set of properties to specify, for example, best and worst case execution time, core affinity, or send and received data. The hardware platform contains information about the number of cores, communication delay parameters, and scheduling options.

In 'Design-Time Performance Analysis of Component-Based Real-Time Systems' [Bon09], Bondarau has the goal of predicting the performance of a system based on the properties of the involved individual components. His *Deep-Compass* framework in conjunction with the *CARAT-RTIE Performance toolkit* [ITE17] is used to create a set of models (e.g. *behavior*, *resource*, or *process*), which can be transformed into executable system models. The *resource* model is used to specify parameter-dependent requirements for software components, like the number of processing cycles. These can either be estimated or measured on a reference processor. The *performance* model is used to model the hardware of the system. This includes – among other parts – processor core, memory, and busses. For each of these elements, various parameters like clock frequency or architecture can be defined. The parameter-dependent specification of the behavior of operations can be modeled with the *behavior* model. The approach does not focus on a target domain or area but provides means to include embedded or industrial automation systems.

Summary:
The related work shows that all of these approaches focus on specific aspects of the automation system. They provide different means to model influence factors, ranging from network specific to cloud-based server farms. They all provide information which can be used to identify important influence factors for this thesis. General (performance) modeling approaches like MARTE [Obj06], Palladio [RBB+11], [Bon09], or [Wan06] are generally applicable. Nonetheless, they reveal (standard) factors that can or must be considered. As mentioned is this list not complete. Depending on the influence factor, additional related work is introduced.

4.2 Parameter Types

Each influence factor can be further characterized by a set of parameters. These parameters vary from factor to factor, but the basic types of parameters are reoccurring. Most of them can be represented by a primitive type like an *Integer* or *Boolean*. But for some cases, it is more convenient to use special types that fit the need of the parameter best. An example is the file size, which includes the size and a selected unit (e.g. kb). Furthermore, there are some parameters that cannot easily be expressed without a complex type. These could be, for example, arrival patterns which specify how often an event occurs. The types defined in this section are a selection or combination of different units and parameters used in the modeling approaches [Dou04, Obj06, Obj05, RBK+07]. They have been selected with a focus on an ease of use and their suitable application in the early development stages of an automation system. The following subsections detail the types used to specify the parameters of the different influence factors in this chapter.

4.2.1 Primitive Types

The most basic type to define a parameter or variable type for an influence factor is the primitive type. Each programming or modeling language defines their own set of primitive types which can be further used. For example, the Unified Modeling Language [Obj15a] specifies primitive domains each containing predefined primitives like *Integer*, *Boolean*, *Real*, *String*, and *Unlimited* (*). In addition to the basic types, UML allows the definition of own types by extending the primitive class. MARTE [Obj06] extends UML primitives by adding *short*, *long*, *unsingedLong*, and many more. The Systems Modeling Language [Obj15b] specifies extensible *PrimitiveValueTypes* and further groups numbers and non-number primitives.

For some influence factors, it is necessary to not only specify how many variables are used but also which type these parameters have. A reasonable example is the *String* variable versus the *Integer* variable. A *String* can vary in length which might lead to a more complex copy process. An *Integer* can be copied and serialized more easily. The OPC server on a ILC 171 ETH2[1], puts a specific load on the CPU depending on the type of variable that is send to a client [FH12, FHMB13].

For this thesis, the following set of primitives is specified. Based on measurements of the firmware during the Phoenix Contact analysis project and related work (e.g. MARTE or UML), this set will be sufficient to describe the different influence factors and their parameters. The Table 4.1 lists these primitives.

Table 4.1: Basic set of primitive types

Name	Description
Integer	Natural numbers (signed)
Double	Double-precision floating-point (signed)
Boolean	Unsigned bit value – false(0) and true(1)
String	Arbitrary long sequence of characters, e.g. ”Hello World”

4.2.2 Filesize

Several services allow the transmission of files over a network like project data or log files. The utilization of the PLC is dependent on the size of the file is processed or sent. To specify a file size, a simple integer would be sufficient, assuming that the unit is fixed to the lowest scale. While this is a feasible approach, it is not a very convenient one. Therefore, most specifications or profiles [Obj05, Obj06, RBB+11] use either predefined units or allow the definition of them. The size of a file can be defined by means of an Integer value

Table 4.2: Units to specify the size of a file

Name	Symbol	Number of Bytes
Byte	Byte	1
Kilobyte	KB	1,024
Megabyte	MB	1,048,576
Gigabyte	GB	1,073,741,824
Terabyte	TB	1,099,511,627,776

[1]PLC - ILC 171 ETH 2TX – `https://www.phoenixcontact.com/online/portal/de?uri=pxc-oc-itemdetail:pid=2700975`

and a unit. The available units are given in Table 4.2. The size of files above Terabyte do not play a vital role in the automation domain and are therefore neglected. However, the list can be extended at any time. Examples of different file size definitions are a log file with 230KB, a configuration file with 12B, and a complete project file (containing settings, programs, and graphics for the HMI) with 3.2MB.

4.2.3 Using Stochastic Parameter: Probability Functions

In some situations, it is better to model certain parameters not with a fixed value, but with a random one. This approach is especially suited for cases in which a program runs smoothly until a certain condition is met and the program execution time peaks. Such a program could run 99% of the time with an average of 6 ms per execution, but in the last 1% a peak will push this time up to 12ms. Setting the 12 ms as a WCET would be a safe approach. But in certain situations exceptions of the usual behavior needs to be considered as well.

In these situations, probability distribution functions can be used. To use stochastic functions, it is first necessary to give a short introduction to random variables. A random variable is a variable whose value can take on a set of possible different values, each with an associated probability [Kol60, All14]. This can also be expressed as a measurable function

$$X : \Omega \mapsto E$$

where Ω is the probability space and E the set of observable events (measurable space). The measurable space is usually mapped to real numbers $\mathbb{R}$. For a dice roll, the random variable X could be 1, 2, 3, 4, 5 or 6. Therefore, the probability space is 1, 2, 3, 4, 5, 6. The probability that X takes value 3 is denoted with $P(X = 3)$ and its value is 1/6. The full notation is P(X = value) = probability of that value, $P(X = 3) = 1/6$. Random variables are needed to specify probability mass and density functions in the following sections.

Providing a simplification for modeling purposes: In the following subsections, the probability mass function and the probability density function are introduced and their usage in this thesis is specified. For certain influence factors, name and notation will be modified to provide a more natural usage. An example for such a simplification is the BoundedExecution (see 4.2.4) which will select a random value between an upper and a lower bound. So instead of providing a distribution function (to a random generator), a simplified notation is used to model the possible values that will be available. Some of these simplifications are inspired by the Palladio Component Framework [RBB$^+$11, Rec08] (see section 2) and by SysML [Obj15b]. A drawback of this approach is that this notation is not as flexible as complex distribution functions like a Bernoulli, Binomial, or Poisson distribution. At the early development stages when multiple factors are only estimated, this less accurate but easier to use notation is sufficient.

Other approaches use a wider range of stochastic functions for continuous values. MARTE provides a set of probability distribution operations including Bernoulli, Binomial, Exp, Gamma, Normal, Poisson and an Uniform Distribution Function. SysML to provides distribution properties via the *distributedProperty* stereotype. This stereotype can further be specialized by a developer. Part of the basic SysML [Obj15b] specification are the interval and a normal distribution as shown in [Wei11].

Probability Mass Function

The first use of a stochastic expression is to define not just one fixed value like the Worst-Case-Execution-Time (see 4.2.4), but a set of possible values. Each value is paired with a probability in form of a double value. Over the complete set of values, the probabilities must add up to exactly 1. This is basically the definition of a Probability Mass Function (PMF) [Ste09]. Figure 4.2 shows two examples. In the left chart (value range example), a PMF is used to select values between 4 and 8 as an execution time depending on their probability. 4 ms and 8 ms have a probability of 10%, 5 and 7 of 20% and the value 6 is used with the highest chance of 40%. In the right chart (peak example), the execution time is 6 ms in 99% of the time. But there are some peaks where the execution time doubles to 12ms. This happens only in 1% of the cases.

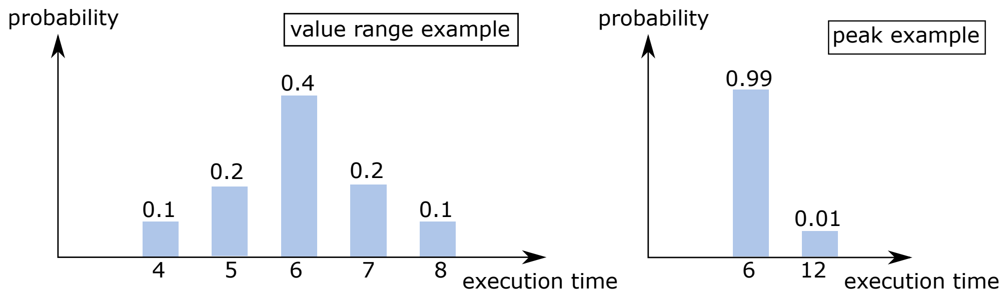


Figure 4.2: Two examples of PMF with different values and their probabilities

To later add these stochastic functions to the Systems Engineering models, a textual representation is needed. Using mathematical formulas or dedicated languages like MathML [ABC+03] to specify the functions would be very time consuming and error prone – especially in the early development phases where exact models for a performance evaluation are not even necessary. Therefore, an easier and more intuitive notation must be found. The Palladio Component Framework [RBB+11] already uses much simpler expressions for modeling of stochastic functions. They defined a full EBNF based language which is capable of specifying the Probability Mass Function in a simple, easy to learn textual expression. Therefore, the language definition used in the Palladio Component Model – more precise in the StoEx model (see [RBB+11, RRMP08] section 2.5 random variables) – is partially used to model all stochastic expressions in this thesis. More information about the Palladio Component Model can be found in Chapter 2. The definition of the PMF provided by Palladio and used in this thesis is:

- IntPMF: A mass function returning different integer values. To describe the two charts of Figure 4.2 in a formal and textual notation, the following two definitions are necessary. For the value range example *IntPMF[(4,0.1) (5,0.2) (6,0.4) (7,0.2) (8,0.1)]* and for the peak example *PMF[(6,0.99) (12,0.01)]*

```
IntPMF[(Int,Double)+]
```

Probability Density Function

To model a range of (continuous) variables, Palladio uses the Probability Density Function (PDF). This basically allows the specification of intervals with probabilities which the random variable falls into. To model the function more easily, the Palladio Component Framework uses a discretization with either fixed or variable intervals. This simplifies the modeling of probability distributions and is especially useful if the function consists of large parts and a few peaks. So instead of defining a PMF for a range from 10 ms to 30 ms with 20 values having the same probability, only one interval needs to be specified.

Intervals with variable length are defined as boxed PDF. The following definition is taken from [RBB+11]. They specify intervals I, so that for each two intervals $J_1, J_2 \in I$, $J_1 \neq J_2$ the disjunction is the empty set $J_1 \cap J_2 = \emptyset$ and the union of all intervals forms a new interval from zero to $x \in \mathbb{R}^+, \cup_{J \in I} = [0, x[$. This means that the intervals do not overlap and that there are no gaps between the intervals. To ensure both properties mentioned above, the intervals are specified by their right hand value only. The result is a set I_X whose values define the right hand sides of all intervals. Now, an order can be defined of these sets set such that $x_1 < x_2 < \cdots < x_{n-1} < x_n$. Then the ith interval is $[x_{i-1}, x_i[$ for $i > 1$ and $[0, x_1[$ for $i = 1$.

Palladio provides just one variant of the PDF.

- DoublePDF: With the textual notation *PDF[(10,0) (30,0.3) (35,0.6) (45,0.1)]* the graph in Figure 4.3 is described. The chance that the random variable takes a value between 0 ms and 10 ms is at 0%, between 10 ms and 30 ms is at 30%. Then there is a peak with a chance of 60% that the random variable falls into the interval between 30 ms and 30 ms. Last, with a chance of 10% the variable has a value between 35 ms and 45 ms. The formal notation for the function is

```
DoublePDF[(Double,Double)+]
```

The Palladio PDF function returns a Double value. For this thesis, each influence factors and its parameters are modeled with a given unit. To simplify the modeling process, the provided values must be each defined as an Integer. This forces developers to model human readable values like 10 min, 1 sec, or 30 Kb, instead of being able to set times to 0,00001 hours. The defined DoublePDF function will also be wrapped by more convenient functions in the following sections, specifying whether to model an execution time or arrival pattern.

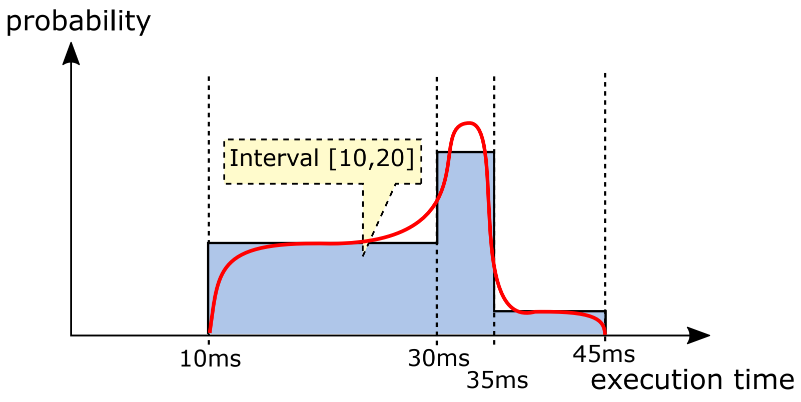


Figure 4.3: A PDF with variables intervals

4.2.4 Execution Time

This subsection focuses on the definition of execution times for Programs, Function Blocks, Functions, or any other part of the automation system that uses a processing resources for a given time. It is often specified with a time value that denotes the overall duration from start to end of an execution. This is a convenient way and usually represents the mindset and experience of the developer best. Dependencies to underlying hardware, parameters, environmental conditions, or other running executions are not considered. Because this value can be roughly estimated, this further simplifies the modeling process especially in the early stages of the system development. This estimation is often based on experience of the developers or the timing behavior is already known from existing software. This leads to a follow-up problem of the transferability of these results to another PLC, which is handled in Chapter 5.

The MARTE UML profile [Obj06] also provides detailed means to specify time and intervals. For this thesis, it is necessary to model how long an execution of code will take. Therefore, the time units that can be used are first set up in Table 4.3 . Times below the nanosecond and above a week have been neglected due to their seldom usage. However, the table can easily be extended in case it is necessary to incorporate such extreme values. Usage examples for time specifications are 100 ms, 1 s, or 100.000 µs.

Most tools and approaches use the worst-case execution time (WCET) [WEE+08, OS97, BB00, EES+02, FHL+01] as a maximum and fixed value the execution

Table 4.3: Available time units

Unit	Symbol	Description
nanosecond	ns	1 second = 1,000,000,000 nanoseconds
microsecond	μs	1 second = 1,000,000 microseconds
millisecond	ms	1 second = 1,000 milliseconds
second	sec	base unit of Time
minute	min	1 minute = 60 seconds
hour	hr	1 hours = 60 minutes
day	d	1 day = 24 hours
week	wk	1 week = 7 days

will take. However, there are more ways to specify the duration of a program execution, like giving an upper and lower bound, or defining a density function. These approaches are explained in more detail. The different ways to model the execution time in this thesis are depicted in Figure 4.4 and will be detailed in the following subsections.

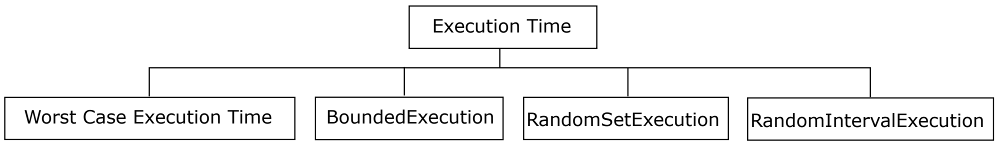


Figure 4.4: The four subtypes to model execution times

Worst Case Execution Time

The Worst-Case Execution Time (WCET) is usually used during the design of real-time systems, where knowing the longest time for an execution of an embedded software is important for its reliability and correct behavior [ZBN93, EES$^+$02, LPT10]. "*WCET analysis computes upper bounds for the execution times of pieces of code for a given application, where the execution time of a piece of code is defined as the time it takes the processor to execute that piece of code*" [PB00]. The WCET is often a pessimistic overestimation of the real WCET, highly processor, architecture, and compiler dependent and in most cases under the assumption that no preemption takes place. Figure 4.5 (modified from [Erm03]), shows the relation between the Best-Case-Execution-Time (BCET) and the WCET. The program execution is finished to a certain probability at a specific time. This time is shown on the x-axis of the graph. If a WCET is estimated to be in the red area, then there are some program executions that take longer then expected – the WCET is wrong. This can lead to severe problems due to false scheduling strategies or the deployment of too many software components on a single PLC. Any WCET that is beyond the red marked area is ok but too high and, therefore, will lead to a waste of resources. The aim of almost all WCET approaches is to get as close to the real boundaries of the program execution as possible.

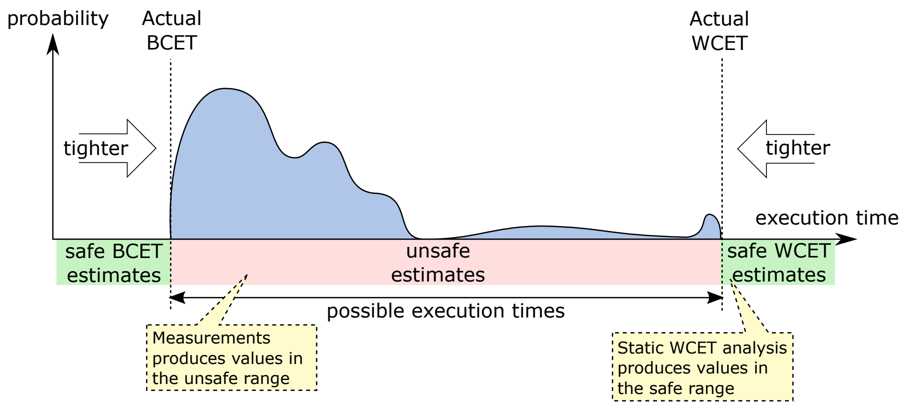


Figure 4.5: Execution time estimates (source [Erm03])

The procedure to determine the WCET varies from approach to approach. The following example is taken from [Erm03]. Figure 4.6 shows the different stages of a program and a WCET analysis. The program is written in source code which is compiled to object code. Running this object code on a hardware allows the determination of the actual WCET. Information provided by the program stages is used for the analysis. Input for a WCET calculation is a flow analysis of the code. Such a flow analysis is often based upon a control flow graph created from the code directly or an intermediate abstract syntax tree. The low-level analysis is used to compute how the blocks in the control flow graph are executed on the target hardware. For modern processors, it is important to consider the effects of performance boosting features, like caches and pipelines.

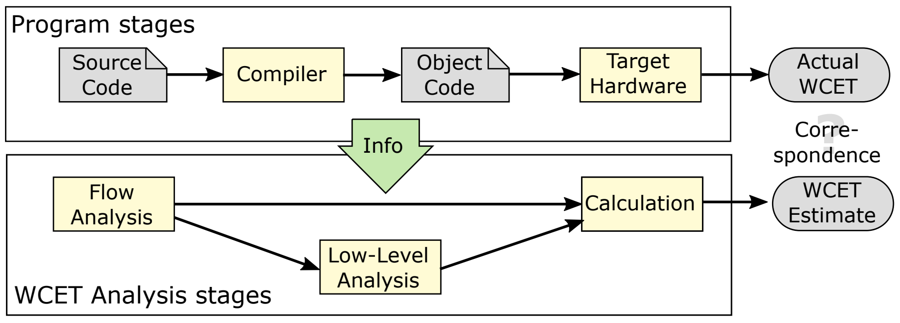


Figure 4.6: Components of a WCET Analysis (source [Erm03])

The different approaches and tools vary from this procedure and make use of a broad range of other techniques like statistical-based estimation [HHM09]. A good overview can be found in [Erm03]. Well-known tools for WCET analysis are AbsInt aiT [FH04], RapiTime Worst-Case Execution Time Analyzer from Rapita Systems [WEE^{+}08], Bound-T Execution Time Analyzer from Tidorum [Tid17], and Chronos an open source static WCET analysis tool from the National University of Singapore [LLMR07].

Of course, there is always the simple solution of measuring a lot of code executions with different sets of input, each time marking the highest time. Another option is using static analysis techniques and count assembler instructions for each function deducing the WCET. But both approaches are quite inaccurate.

BoundedExecution

With the bounded execution, a lower and upper bound are set in which the run time of a program may vary. Basically, the best and worst execution times are fixed and the values in between are uniformly distributed. The *BoundedExecution* function returns an integer value. Figure 4.7 illustrates the possible execution times in the blue area which is enclosed by the WCET and the BCET.

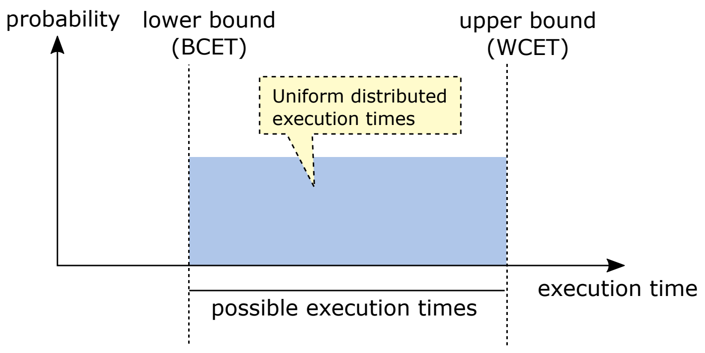


Figure 4.7: A uniform distribution between lower and upper bound

For returning discrete values, the BoundedExecution function can be realized by a Probability Mass Function (PMF) (see 4.2.3) which returns a random variable that is equally likely to take any of the integer values i_{min} and i_{max}. The lower and upper bounds are used to calculate the interval length. Based on the length, the probability for each discrete value (integer) can be calculated and used for the PMF. For a simple and intuitive usage during the development of automation systems, this way of modeling provides a more detailed insight to the execution of a program. For this thesis, the execution time is specified as a discrete Integer value and each specification must include a time unit. The time unit should be on a fitting granularity but can be freely chosen by the developers. When setting up a bounded execution, the function should return values as discrete numbers (Integer). Note that the definition of bounded execution times will lead to more precise performance evaluation results compared to the fixed WCET. But to be safe that a deployment of software or scheduling strategy will work with the chosen automation system design, worst case execution times should be preferred over bounded executions. The simplification of the PMF narrows the return values down to just integer values. This is feasible because the according time unit is provided.

The notation to specify these values is given below:

```
BoundedExecution[(Integer:Integer),Timeunit]
```

An example for the specification of an execution time for a Program or Function Block that lies between 10 ms and 15 ms could be done with *BoundedExecution[(10:15),ms]*.

RandomSetExecution

To model a set of execution times and assign each value a certain probability, the Probability Mass Function (see 4.2.3) can be used. The *RandomSetExecution* wraps the access to the PMF function to simplify its usage and to indicate that the returned values are used for execution times. It does not add any additional information to the PMF function. The parameters of this function are tuples containing an execution time value and its probability formulated as a Double value ranging from 0 to 1. The sum of all probability values must be exactly 1.

```
RandomSetExecution[(Int, Double)],Timeunit
```

RandomIntervalExecution

Similar to the *RandomSetExecution* the *RandomIntervalExecution* wraps the PDF function into a more convenient representation for the developer. Its parameters are tuples of ranges and probability. The range is set as two integer values, denoting the upper and lower value. The probability is formulated as a Double value, also ranging from 0 to 1 and with an accumulated value of exactly 1. During the analysis, the *RandomIntervalExecution* will provide integer values randomly chosen from the given intervals. Therefore the interval limits can be specified without the use of including ("20]") or excluding ("20[") boundaries, as it is done for interval definitions handling non-natural values. It is assumed that the random values are uniformly distributed. A usage example is *RandomIntervalExecution[([10,15],0,3) ([16,20],0.2) ([21,25],0.5)]*. This will return an execution time from 10 to 15 in 30%, from 16 to 20 in 20%, and from 21 to 25 in 50% of the cases.

```
RandomIntervalExecution[([Int,Int], Double)+],Timeunit
```

4.2.5 Access Frequency / Arrival Pattern

Each influence factor puts a load onto the PLC increasing the overall utilization. This effect can happen continuously in the background or if a certain action is performed. Function Blocks and Programs, for example, are executed when its containing Task is triggered. The Task execution, however, is set to a specific interval or event. The same applies to services that can be accessed by external systems like a management server or HMI. The HMI can be configured to grab all necessary variables from the PLC in a specified refresh interval of the web page. Or it could request the variables only if a user interacts with it. Therefore, two different arrival patterns are usually defined: *periodic* and *aperiodic*. The first is used when a fixed interval can be set between to sequenced events. If this is not possible, an aperiodic pattern must be used. This can further be detailed as bounded, bursty, irregular, and stochastic, like defined in [Dou04]. Bounded is used when a minimum and maximum arrival time can be specified. Bursty describes patterns with a high change of events in a short interval. To model a bursty pattern, a maximum burst length and a burst interval must be set. Irregular means that the arrival times can not be specified. And the stochastic pattern is used if the value can be specified by a probability function. Other modeling languages use similar approaches to model these two types. MARTE [Obj06] introduces *ArrivalPattern* and distinguishes between *PeriodicPattern* and *AperiodicPattern*. AperiodicPattern can further be categorized into *SporadicPattern*, *BurstPattern*, *IrregularPattern*, and *ClosedPattern*. The UML profile for Schedulability, Performance, and Time Specification [Obj05] uses similar patterns names *bounded*, *bursty*, *irregular*, *periodic* and *unbounded*.

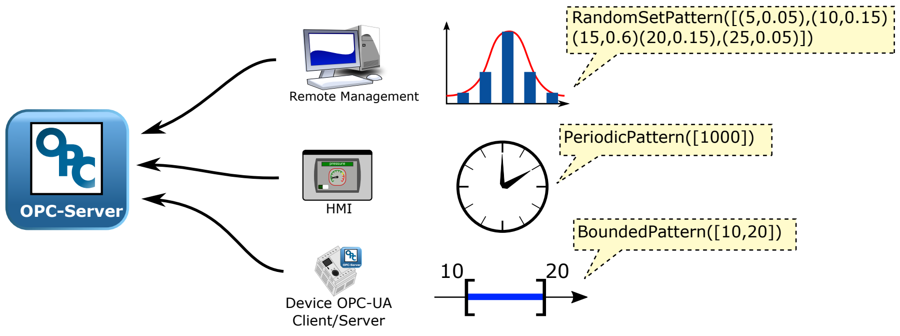

RandomSetPattern([(5,0.05),(10,0.15)(15,0.6)(20,0.15),(25,0.05)])
Remote Management
PeriodicPattern([1000])
OPC-Server
HMI
BoundedPattern([10,20])
10
20
Device OPC-UA Client/Server

Figure 4.8: Examples for different access frequencies

For the specification of influence factors in the early development of automation systems, it is best to focus on an easy rather than precise modeling. This is often not even possible due to the missing information in this stage of the development. Therefore, only three different arrival patterns are used in this thesis. Figure 4.8 depicts the three kinds.

Figure 4.9 gives an overview over the different access/arrival patterns that are used in this thesis. In addition to the periodic access, three more stochastic access pattern are used. Each pattern is briefly described in the following subsections.

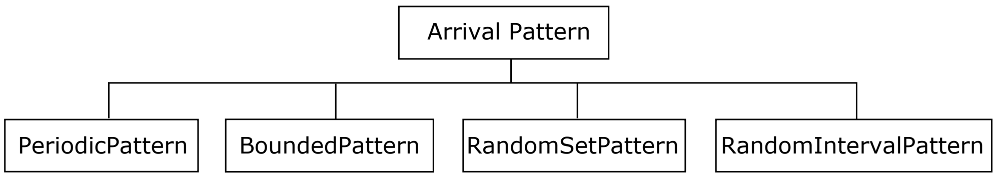


Figure 4.9: Different subtypes of the arrival pattern

PeriodicPattern

The fixed or periodic pattern is a common specification for technical systems that have a fixed refresh interval like. This pattern can be used for a SCADA system which collects data from a PLC in fixed intervals. Profiles like MARTE [Obj06] provide even more details by allowing the developer to set additional parameters like jitter. Such a jitter might be induced internal scheduling or network-related delays. However, these fine-grained details are neglected here but can be added later by providing additional properties to the pattern. The access in fixed intervals can be specified with the notation given below. An example of the usage is *PeriodicPattern([1000])*, triggering a utilization or service access in intervals of 1000 ms.

```
PeriodicPattern[Integer]
```

BoundedPattern

Instead of setting a fixed interval, the *BoundedPattern* allows providing an upper and lower limit from which a value is randomly selected. This behavior is similar to the *BoundedExecutionTime*. The function needs two parameters as the two limits and will return an Integer uniformly distributed between the two given limits. This enables a developer to set the time between two events to a range of values instead of a fixed one. The bounded access can be specified with the following notation.

```
BoundedPattern[Integer:Integer]
```

The example shown in Figure 4.8 could be specified with *BoundedPattern([10,20])*, using random intervals between the bounds 10 and 20.

RandomSetPattern

The *RandomSetPattern* is used to wrap the Integer based Probability Mass Function into a dedicated function that provides a randomly selected value. The RandomSetPattern accepts tuples setting the integer value and its probability as a double, ranging from 0 to 1. The sum of all probabilities must be exactly 1. The RandomSetPattern will return an Integer value based on these probabilities as described in 4.2.3. The example given in 4.8 shows the definition of a random set with the notation *RandomSetPattern([(5,0.05) (10,0.15) (15,0.6) (20,0.15) (25,0.05)])*. The peak value of 20 and therefore the time between two successive events will be selected with a probability of 60%.

```
RandomSetPattern[(Int,Double)+]
```

RandomIntervalPattern

The last arrival pattern is the *RandomIntervalPattern*. Like the *RandomSetPattern*, is it similar to its execution time counterpart (RandomIntervalExecution) and just simplifies the usage. With the expression *RandomIntervalPattern[([10,15],0,3) ([16,20],0.2) ([21,25],0.5)]* three intervals are specified. An evaluation of this expression during an analysis will return a value from 10 to 15 in 30%, from 16 to 20 in 20%, and from 21 to 25 in 50% of the cases. The RandomIntervalPattern will return Interger values. Therefore, the interval limits can be specified without the use of including ("20]") or excluding ("20[") boundaries.

```
RandomIntervalPattern[([Int,Int], Double)+]
```

4.2.6 Operations

Some services and Functions Blocks allow the execution of operations they (or other components) provide. To model these operations a simple string can be used to specify which operation should be triggered. However, the OPC-UA server, for example, allows the execution (call) of methods with a set of parameters and return values. To be able to later specify the influence factors up to a fitting level of detail, the different method parameters need to be considered. The return values of methods complicate the modeling of influence factors. Additionally, they will most likely not be used as an input for further computations as the underlying analysis tool needs to support dynamic variables and variable binding. The operations defined in this model are used to invoke or simulate a certain utilization at the PLC. A 100% representation of real code (and functions) is not necessary. Therefore, the type Operation is defined as a set of parameters and no return value. Each parameter is a tuple of name and type, referring a primitive type as specified in subsection primitive types (4.2.1). In this thesis the following syntax is used:

```
OPERATION = opName "(" PARAMTER* ")";
PARAMTER = (variable:TYPE);
TYPE = ("STRING"|"BOOL"|"INT"|"DOUBLE");
```

Examples are *updateCache()*, *writeValues(value:INT)*, *getUserList()*, or *authenticateUser(username:STRING,password:STRING)*.

The detailed definition of operations indicates an in-depth modeling of influence factors or services. It should be checked, whether this level of detail is necessary for the desired goal of conducting an early evaluation of the automation systems performance. Often, simpler definitions like an average base load or a varying execution time could be used instead.

4.3 Applications

The category *Applications* groups all kinds of compiled or interpreted code and their execution configurations that are used to fulfill the systems function. Applications do not include PLC specific executables as part of the firmware or services. For the milling example introduced in Chapter 3, typical programs control the robot movement, the workpiece delivery, or the engraving station. A list of programs used in the example is given in Appendix A. In addition, to control programs, other functions like writing log files in specific time intervals or the sending of TCP/IP messages to a remote server could be realized by dedicated programs.

In the IEC 61131-3 standard [Int13a] Programs, Functions, and Function Blocks are grouped under the term Program Organisation Unit (POU). Functions can be assigned parameters, but have no state and static variables. The Function Block (short FB) has input and output parameters and static variables. A very important property of the FB is, that they keep their internal state after they have been instantiated. So the output of an FB depends on its current state. The Program represents a main part of an application and is used (beside the resource) for global variables and the assignment of physical addresses. In all other aspects, the Program behaves just like a Function Block. Both Programs and Function Blocks are executed/triggered by a task. More details on POUs, resources, and configurations can be found in chapter 2.

Figure 4.10 shows the software model defined by the IEC 61131-3 standard and all the previously introduced POUs. It depicts the configuration as the main container in which one or more resources can be defined. Each resource handles the different Task types which are used to execute the associated Programs and Function Blocks. Function Blocks can be instantiated and accessed inside a Program but may be triggered by a completely different task in the same resource. Variables can be accessed via a global scheme to allow Programs and Function Blocks to exchange information.

Programs, Function Blocks, and Functions are part of the influence factors that can be specified in the early development stages. In case the automation system use existing POUs or is a modified version of an older system, this in-depth knowledge is already available. Otherwise is it easily possible to estimate Program execution times and refine them with Function Blocks and Functions in the later iterations of the development. Programs and Function Blocks can usually

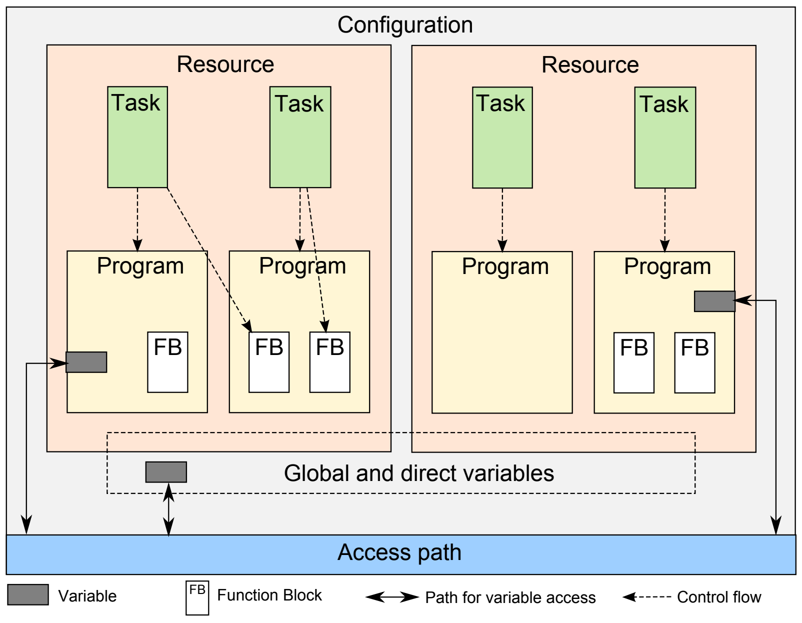


Figure 4.10: IEC 61131-3 software model, modified from [Int13a]

be mapped to specific functions or modules of the automation system. Another reason why a coarse model of the automation systems software is usually available in the early system design phases [Dub11].

4.3.1 Programs

There are different kinds of programs used to realize the system's desired functionality and behavior based on the capabilities of the used PLCs. These kinds can be roughly grouped into IEC based programs and native executables. The majority of programs can be assigned to the first group, the IEC based programs.

The International Electrotechnical Commission (IEC) [Int16a] has defined a worldwide standard for the programming of PLCs [Int13a]. The different parts of this standard and its accompanying programming languages are described in detail in Section 2.1.1. Programs are the top level elements which can be used to structure the application. Each Program is written with one of the five IEC languages and executed via tasks. A Program can contain Function or Function Block calls (see 2.1.1), but cannot execute other Programs.

IEC Programs are usually programed with a vendor specific engineering tool, whereas not all vendors support all five IEC languages. Some vendors even extend the existing language capabilities or develop entirely new ones (S7 SCL (Structured Control Language) [SIEc], S7-Graph [SIEb] or SIMATIC S7 CFC (Continuous Function Chart) [SIEa]). A current trend is to add model-driven approaches to the engineering tools to cope with the increasing complexity of automated systems. State machines or other UML [Obj15a] like diagrams

[Bec16b, Smab] can be used to model the programs structure and behavior. Code generators will transform these models to IEC languages like Structured Text, which can be easily edited afterwards.

To execute the programs they usually need to be compiled for the runtime environment (see 2.1.1) of the PLC. This runtime environment is used (among various other functions) to set up the tasks, global variables, and to provide access to the IO variables from the IO system.

Currently, the minority of programs executed on PLCs are native code based. Native programs are compiled for a specific PLC and its architecture. The various vendors of PLCs each support a number of different programming languages. The most common are C, C ++ and C#. Special frameworks or libraries need to be included in the native code programs to access the PLCs runtime environment or its equivalent APIs. However, in the last couple of years, model-driven tools such as Matlab/SIMULINK became more and more popular for the design of automation systems behavior, increasing the use of native programs. “Simulink is a block diagram environment for multidomain simulation and Model-Based Design. It supports simulation, automatic code generation, and continuous test and verification of embedded systems.” [Mat16, Mat17]. Matlab/SIMULINK simplifies the design of control engineering tasks and is far more intuitive for the control engineer. Matlab/SIMULINK offers the option to easily add new functionality for code generation of specific targets (PLCs) and compile the code in a single workflow. This allows the engineer to deploy the code directly onto the PLC without further changes or working steps. If there is no code generator addon for a specific PLC available, Matlab/SIMULINK can also generate plain C or C++ code which could be manually compiled and deployed onto the PLC.

Despite the differences between IEC and native based programs, they share common properties. Running code on a PLC will put a certain load onto the CPU. This utilization is highly dependent on the execution time for each Program. There are different ways to model the execution time of a Program as specified in Section 4.2.4. A second parameter that needs to be considered is the frequency in which a Program is executed. In more sophisticated systems multiple cores are available that allow the parallel execution of Programs. Depending on the vendor and PLC is the developer allowed to change core settings to fine tune the system. Parameters of the influence factor Program can be defined as follows:

4.3.2 Function Blocks

The IEC 61131-3 Function Block (FB) can be compared to a Program. An FB must be executed within a Task or Program and keeps it internal variables after initialization and execution. A Function Block can have both input and output parameters and must be instantiated in a containing Program or FB. Several basic types are specified in the IEC 61131-3 and the PLCOpen standard. This

Table 4.4: Influence factor parameter for Programs

Name	Type	Description
Execution time	ExecutionTime	The time it takes the Program to run through the code for one execution without preemption
Frequency	ArrivalPattern	The frequency in which the Program is executed (by a task).
Core	Integer	Optional parameter specifying the core on which the program is executed on. Usually the operating system scheduler handles this assignment (see section 4.4.2)

common set of types is usually extended by vendor specific Function Blocks. Figure 4.11 shows three different Function Blocks. The *CTUD* block is used as an up/down counter and specified in the IEC 61131-3 standard. The second Function Block is UA_connect which is used to create a (secure) transport connection and an OPC-UA session. This connection must be terminated by another FB (*UA_Disconnect*). The last FB is provided by Phoenix Contact as a PLC vendor. The *DBFL_MySQL_ACCESS* block allows access to a database which is stored on a MySQL server. The data types of the in and out parameters are omitted in the figure.

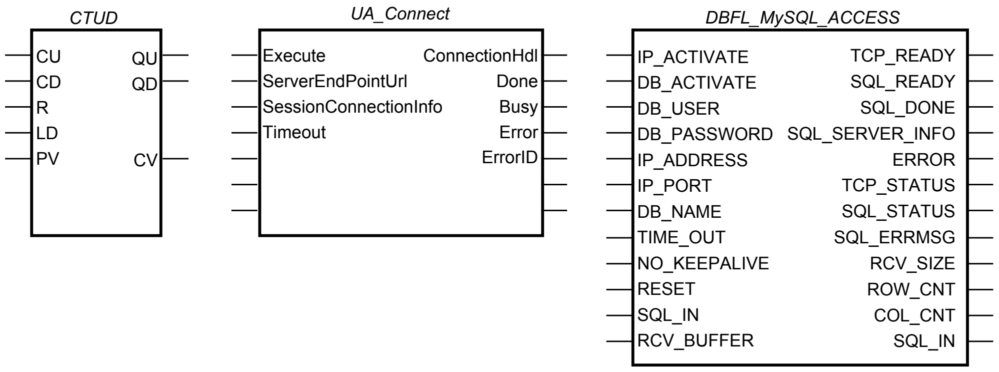


Figure 4.11: Three Function Blocks examples: IEC 61131-3 standard, PLCOpen standard and a vendor specific.

Another example for a vendor specific Function Block is an IP-Block which allows the developer to open a IP Communication via UDP or TCP protocol. This example shows perfectly, that some of the Function Blocks have a close integration into the underlying operating system. So invoking a Function Block and executing its code often triggers firmware specific operations. These operations might also be running in parallel to the IEC Programs und Function Blocks, leading to a background or base utilization of the PLC. This background utilization (base load) can be slightly different on PLCs using the same

IP Function Blocks but varying firmware version or operating systems. Therefore it is necessary to measure the usage of the Function Block for each PLC it will be executed on.

The in and out variables of a Function Block are used to set the parameters and (indirectly) select one of possibly multiple functions the FB can perform. To specify the different utilizations of these functions, it is best not to model each in and out parameter but to use a higher abstraction level by just defining abstract operations. For the *IP-Block* this are for example *connect*, *send*, *receive* instead of setting the in ports to specific integer values.

This does not only simplify the specification process, but it is also more intuitive for the developer and allows an easier setup of execution probabilities for each function. It is also tailored towards the increasing use of object orientation in the automation domain [Vya13]. Since the third edition of the IEC standard, Function Blocks can also explicitly specify methods, use inheritance, and interfaces. See [Wer09] for more information. This further supports the decision to abstract the functionality of a Function Block into different provided operations.

The Table 4.5 lists all parameters needed to model the influence factor Function Block. Currently, the definition of operations and their according utilization is separated. In the formal model introduced in Chapter 5 a more condensed (and natural) notation will be introduced.

Table 4.5: Influence factor parameter for Function Blocks

Name	Type	Description
Baseload	ExecutionTime	A background noise that is always induced by the Function Block. This parameter is optional and applies not to all FB.
Frequency	ArrivalPattern	The frequency in which the Function Block is executed
Operations	Operation	This parameter models the different operations that abstract the behavior of the Function Block.
Load per Operation	ExecutionTime	This parameter must be specified for each Operation. It defines the execution time for each operation call.

4.3.3 Functions

Functions can be specified and called from Programs, Function Blocks and other Functions. They do not hold a state between calls – which is one of the main differences to the Function Blocks – and do not need to be instantiated. Each Function provides a return value upon execution and can be further detailed by specifying zero or more parameters. Their influence on the PLC performance is given by the parameters and the rate in which they are called. Like the definition of Operations the return values of Functions are neglected. These return values complicate the modeling process and will most likely not be used as an input for further computations as the underlying analysis tool needs to support variable binding. Functions usually provide detailed and specific operations (on data) and therefore should not be extensively used in an influence model tailored towards a high level Systems Engineering model. The following table identifies the frequency a function is called via an arrival pattern. However, in most cases Functions are called during the execution of a Program or Function Block and therefore do not necessarily require an ArrivalPattern. The definition of a Function is covered by the use of the previously introduced Operations. The table of parameters contains just the frequency in which the functions are called and a definition of the execution time per operation. The formal model introduced in Chapter 5 provides a more condensed (and natural) notation.

Table 4.6: Influence factor parameter for Functions

Name	Type	Description
Frequency	ArrivalPattern	The frequency in which the Function is called
Load per Call	ExecutionTime	This parameter must be specified for each Operation. It defines the execution time for each operation call. It depends on optional parameters.

4.3.4 Tasks

To calculate or simulate the utilization of a PLC, it is important to know exactly when and what Programs, Function Blocks, or Functions are executed. This information is given in the task configuration. There are several types of Tasks specified in the IEC 61131-3 [Int13a] standard (see chapter 2), each with its own timing and execution semantics that can be set up in the task configuration.

If tasks are executed too often (oversampling) or with overlapping intervals, the available resources of the PLC might not be sufficient to execute them in a timely manner. Each part of an automation system has different requirements for the execution of its software components. Connections to external systems, internal program timeouts, safety components, and of course motion control with fast access to the IO system are just a few examples of these requirements. Therefore, it is vital to set up fitting tasks and their associated Programs and Function Blocks. Tasks also help to further structure the overall application and separate the different controlling aspects.

A common parameter for each Task type is the priority. This priority is used by the operating system to execute tasks with a higher priority before processing the lower ones. The scheduling influence factor is detailed in Paragraph 4.4.2, the parameter, however, is defined at the task. In following subsections detail the different types of tasks and their individual parameters.

Cyclic Task

All Programs and Function Block instances associated to the cyclic task will be executed in a fixed time interval. Figure 4.12 shows a graph with two programs executed in one task. The programs vary in their execution time, but the interval of 10 ms is fixed resulting in a varying delta time between the task trigger events.

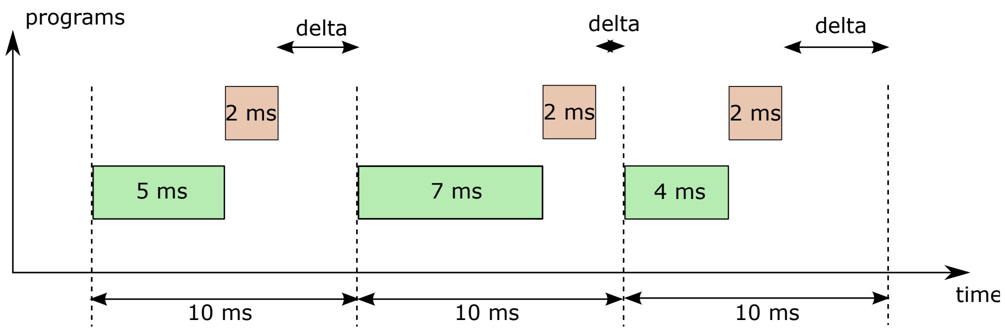


Figure 4.12: Execution of a cyclic task with a fixed interval time of 10ms

In addition to the derived attributes from the (abstract) task, Cyclic Tasks contain a priority attribute for handling the execution ordering of multiple tasks at the same time point or for preemption purposes. The Figure 4.13 shows three different Cyclic Tasks with varying intervals and priorities. The associated Function Blocks and Programs are not shown here. The first task (*task_hmi*) is executed every second and is used to gather data for the graphical interface. *task_wpd* is used to handle all programs for the workpiece delivery component and is executed every 50 ms with a higher priority than the less important GUI task. For motion control, very fast cycle times are needed to achieve the desired precision of the tools. Therefore, *task_engraving* is executed every 5 ms and with an even higher priority than all other tasks. If two or more tasks are executed at the same time point, the tasks with the higher priority are executed first, followed by the ones with lower priority. If a high priority task is triggered while a low priority task is running, this task will be paused and after the execution resumed. To model the influence factor CyclicTask (see Table 4.7) only two parameters needed. First the access frequency to specify the time interval in which the task is triggered and the priority of the task. Every other aspect that influences the utilization or behavior of the PLC can either be neglected or is covered by other influence factors like the firmware (see 4.4.3).

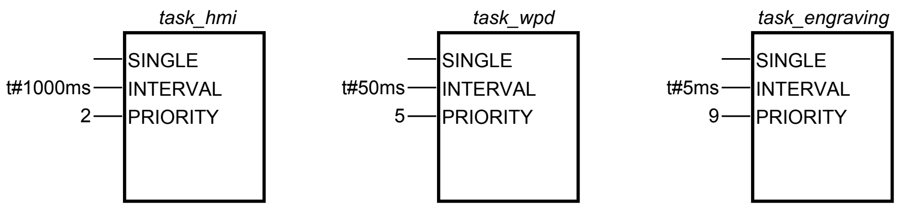


Figure 4.13: Cyclic Tasks with different priorities and intervals

Table 4.7: Parameter for the influence factor CyclicTask

Name	Type	Description
Frequency	Cyclic	The frequency in which the task and its associated POUs are triggered
Priority	Integer	Priority of the task

Event Task

The second task type is the *triggered task* or *event task*. An external trigger (e.g. a rising signal) is used to start the task and, therefore, execute all Programs and Function Blocks of the task. The sources of this trigger event could be external systems (e.g. SCADA) setting remotely a variable, an input from a light barrier sensor transported by the fieldbus, or other programs that are running on the same PLC. Figure 4.14 shows the execution of an event task, denoted with the event sign in the bottom part of the figure. The intervals between the program executions as well as the delta times vary.

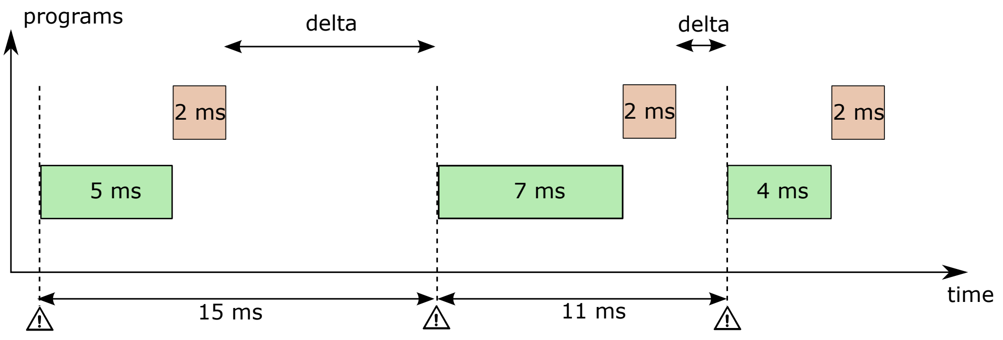


Figure 4.14: Triggered EventTask with varying delta times

The Event Task must be considered for the development and performance prediction of an automated system. However, the modeling and analysis of different events, event sources, and relations between events get complicated fast. For an easier handling (especially in the early development phases) it is, therefore, best to abstract from the event sources and focus just on the probabilities of an event to occur. The Table 4.8, therefore, contains just the parameter access frequency with the two stochastic methods (PDF and PMF) and the priority of the task. MARTE [Obj06] (see Section 2.3.1) uses a similar approach where different timer (schedulable resources) can be further annotated by arrival patterns.

Table 4.8: Parameter for the influence factor Event Task

Name	Type	Description
Frequency	ArrivalPattern	The frequency in which the task (and its associated POUs) is triggered
Priority	Integer	Priority of the task

Idle Task

A subtype of the Event Task is the *Default*, *Free Wheeling* or *Idle Task*. This task can be best compared to an endless loop in which all its associated Programs and Function Blocks are executed - the triggering event is always set to true. Depending on the vendor of the PLC, the loop does not start right after its last execution but waits for a specific time (e. g. a fixed 4 ms delay, 10% of the tasks execution time or using a specific formula). Idle tasks are usually selected for all Programs that should be executed as often as possible and where the exact timing is not important. The “cycle time” of the Idle Task is the added execution time of all associated Programs and Function Blocks, which results in a jittering execution and utilization. The Figure 4.15 shows the execution of two programs. The delta time – or in this case wait time – is calculated every cycle based on different vendor specific settings. In this case, the delta time is calculated with

$$\delta = \left\lceil \frac{\sum_{i \in Programs} exectime(i)}{2} \right\rceil$$

which is added on top of the execution time. This approach is often used to wait for other low priority tasks like the Webserver (see services section) to get some time to execute.

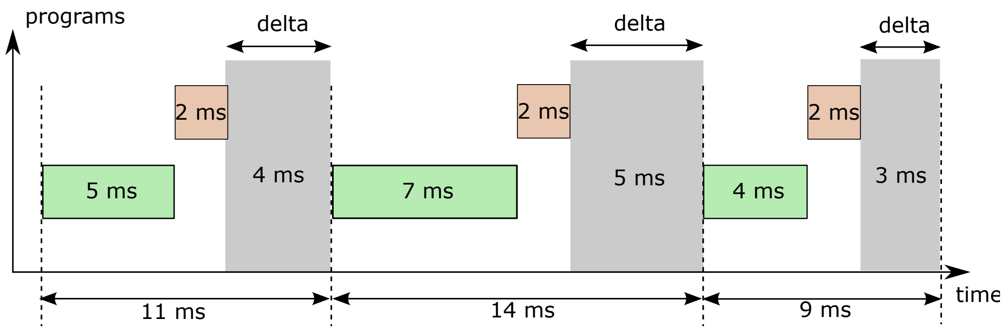


Figure 4.15: Wait times (delta) based on cumulative execution times

The Idle Task is, therefore, highly dynamically and it is best to not specify the access frequency of the idle task as the sum of all contained Programs and

Function Blocks. For this reason, the Idle Task must be incorporated into the influence factors as a dedicated task type. An subsequent analysis must reflect this behavior and consider different execution times and preemption. In Chapter 6 the Palladio Component Framework has been selected as a simulation framework and its internal scheduler extended to allow use of Idle Tasks.

The only parameter that can be set for the Idle Task is the inherited priority. However, most vendors do not allow to manually set the priority of this task due to unforeseen interactions with other tasks and processes. The following Table 4.9 lists the parameters.

Table 4.9: Parameter for the influence factor Idle Task

Name	Type	Description
Priority	Integer	Priority of the task

4.4 PLC

In this subsection vendor specific and PLC dependent influence factors are considered. They include all hardware and software related factors like the CPU, architecture, memory, or software (=firmware). Some of these factors must, others should be to some degree considered when conducting a performance prediction.

Most of the factors identified in this subsection impact the performance of a PLC only slightly or require in-depth knowledge of the PLC, its structure, and behavior. These factors and their details should be hidden in the corresponding analysis/simulation models and not be considered in high-level Systems Engineering models. Therefore, this information needs to be wrapped into external definitions. In this thesis, the term *load profile* is used to model specific behavior of an influence factor based on its parameters. The PLC is an ideal example for a load profile. It hides detailed information form the developer like an operating system background utilization. Analysis tools and models can load profiles and use the detailed information. Load profiles can also be used for various other influence factors, like an FTPServer.

4.4.1 CPU & Architecture

The most obvious influence factor that impacts the utilization of a PLC is the underlying computational hardware - the CPU. PLC vendors use various types of CPUs, each with different architectures (e.g. x86, ARM, PowerPC), cache sizes, instruction sets, and a varying number of cores. All these factors do influence the actual computation power of the PLC and subsequent the utilization during operation. Even different compilers and compiler settings for a specific CPU/architecture can have a measurable influence. The most important CPU related factors are listed below, each with a short explanation how they influence the performance prediction and to what degree. However, most of these factors are neglected due to the high amount of modeling effort, the need to have existing source code or to the – in relationship to the other factors – far to low impact on the performance. Noteworthy for the early prediction of PLC utilization are just two factors: CPU frequency and the number of cores. These are also the two main factors that are used in a wide range of performance analysis tools with a focus on high level functions of a system. Examples are [HCÅ03], [BKR09], [FL09], or [Kah01]. As stated by [Jai90], it is necessary to remain on an appropriate modeling level for the desired prediction goal. In comparison to the other factors, going too much into detail will not yield a way more accurate prediction of the systems utilization.

- **Cache and Caching:** Each CPU and PLC board uses a different amount of first, second and third level cache. This cache has a huge influence on the overall execution of programs due to the fast access on already loaded information [LL99, MB91, LM12, WSR99, LMW99]. Each *cache miss* - data that is not in the cache and must be loaded from RAM (Random Access Memory) or even the hard drive - causes long loading times [WM95, GMM97, DM97]. To incorporate a cache into an analysis is a complex task. The size of each cache and the different compiler optimizations must be taken into account. To calculate the correct execution times of a program, the source code must be also available – which is for an early performance prediction usually not the case. Last but not least, every program that is running on the CPU affects the caching behavior. So the holistic system, including all processes and threads, must be considered. Therefore, it is not feasible to include the runtime improvements of caching into the early performance prediction of an automation system. As for the use of higher level caches (third level memory – RAM) it is safe to assume that for future applications and PLC generations the size of the RAM is always sufficient.

- **Compiler:** Modern compilers use a wide range of optimizations to generate machine code perfectly optimized for a specific CPU and architecture. Turning on optimization flags for example, makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to later debug the program [CFA$^+$07, HKW05, GH01, Han13]. Developers still have the possibility to influence these optimizations as they see fit. As stated before, to make predictions about the improvements of different compiler settings, the source code of a program must be available. It is, therefore, a too complex and time-consuming task to consider the influence of different compilers and compiler settings in the early development stages of an automated system. Additionally, in comparison to the fixed, compiled code of the firmware and operating system to the varying applications in form of Programs, Functions and Function Blocks this factor can be neglected. They are covered by the base utilization of the operating system (4.4.2).

- **Architecture:** Running a program on different architectures (with varying instruction sets) can also influence the execution speed of a program [GH01, LM12]. This effect is not as huge as cache misses but depending on the computational tasks like plain calculations (number crunching) or image processing, this could be a measurable factor. Modern CPUs also use *Pipelining* [SL05] to execute more instructions per clock interval, increasing the performance of the CPU. Instead of processing each instruction sequentially, each instruction is split up into a sequence of steps and each step can be executed in parallel. This effect is also dependent on the compiler and its settings, ordering or using instructions that can be best split up into steps. However, similar to the compiler and compiler settings are the effects of different instructions sets and pipelining hard to model with and without the source code. To still take performance improvements of pipelining into account, one option is to use a modifier/-multiplier on the CPU frequency. More on this workaround can be found in the descriotion of the factors *CPU frequency* and *Multicore*.

- **Additional Hardware:** PLC vendors use in conjunction to a CPU often other computation devices for specific purposes like communication. Complex Programmable Logic Device (CPLD), micro controller units (MCU), digital signal processor (DSP), graphics processing unit (GPU) or field-programmable gate arrays (FPGA) are examples for these kinds of devices which can be integrated into the base board of the PLC. They all can achieve outstanding performance for their specific tasks [AMY09]. For PLCs, the FPGA is often used to handle the high performance fieldbus communication – it is necessary to send and receive huge amounts of data from multiple (io)devices in specific time intervals with jittering under just a few µs. If PLCs provide special hardware, this should to be considered in the performance prediction. However, the modeling effort to incorporate different processing resources is very high. Programs and Function Blocks will be to large parts executed on the CPU. In contrast, specific function calls might be redirected to an FPGA and therefore must be modeld explicitly and assigned to the FPGA. This additional modeling effort is inadequate in the early development stages. Additionally, do most of the vendors use the hardware for tasks like regular network communication, special IO communication, or other operating system related tasks. Therefore it is better to ignore special hardware for the Program and Function Block analysis and consider the influence of hardware in the factors Operating system (Section 4.4.2), IPTraffic (Section 4.4.4) and IO (Section 4.5).

- **CPU frequency:** The clock frequency (or clock rate) of the central processing unit is the most obvious factor to value/rate the performance. An internal clock regulates the rate at which instructions are executed – the more instructions per second can be executed, the faster a program is started, run, and finished. The clock frequency is measured in hertz (Hz). Other factors influence the overall computational power of modern CPUs as well. Technical limits inhibit to go for higher clock frequencies and force the vendors to implement new technologies like complex pipelining or multicore architectures [Man00, Gee05]. Still, to compare program executions – respectively predict their execution – on different CPUs, one of two common attributes that can be used is the raw clock frequency.

- **Multicore:** The second attribute that is very important for the performance prediction of PLCs is the number of cores (or CPUs). Multicore architectures were very uncommon in the automation domain for the last years but due to the recent drop in availability of single core CPUs, they become more and more popular. One reason why multicore CPUs were avoided is the non-determinism that accomplishes the distribution of threads and process onto different cores and the resulting parallelism during the program execution. Like caching, it is more complex to ensure the flawless execution of firmware and application code. This is especially important for safety critical applications and PLCs that must be certified for these kinds of usages. But still, multicore CPUs provide a huge boost in performance, under the premise that the programs can be executed in parallel [Gee05]. For the performance prediction of automation systems, not just applications (Programs and Function Blocks) but also operating system functions and additional services must be considered. They provide a rich and easy to parallelize environment that can make use of the multicore architectures and the resulting performance boost. Shared resources that could influence the parallel execution are foremost the network communication and the fieldbus data exchange. Via configurations are concurrent accesses reduced to a minimum (process data mapping) or handled down to special hardware as mention in the *additional hardware* factor. Therefore, it is possible to - at least for an early performance prediction – neglect waiting times between parallel processes.

The Table 4.10 lists the minimal amount of parameters that must be considered when modeling the hardware of a PLC.

Table 4.10: Parameter for the influence factor PLC

Name	Type	Description
PLC	String	Name or type of PLC that will be used
Number of Cores	Integer	The number of available cores for parallel execution of code
CPU Frequency	Integer	The frequency of the processor as an abstract unit for instruction throughput

4.4.2 Operating System

Almost all modern PLCs use off-the-shelf (real-time) operating systems [TWTT87, SGGS98] under their custom firmware. Depending on the desired use of the

PLC (e.g. motion control or safety), the different operating systems each provide unique perks. Common choices for such operating systems are for example embOS [SEG], Windows Embedded [Mic16], VxWorks [Win], QNX [QNX16], Real-time Linux (CONFIG_RT_PREEMPT [Lin16] or RTAI [RTA16]), FreeRTOS [Fre16] or Nucleus OS [Men16].

Each operating system has different architectures and background processes with their unique programs and performance profiles [RBH$^+$95, WSR99]. In addition, they can handle different tasks (like network communication) much more efficient than others. For a performance prediction of a PLC, it is, therefore, necessary to know the underlying operating system and its utilization under different working conditions. Capturing each individual process and its behavior for certain tasks could lead to a highly detailed model of the PLC and therefore to a much more precise prediction. As for the early design evaluation of automated systems – where the software and detailed tasks might not even exist yet – the modeling and consideration of a simpler background 'noise' should be sufficient. Therefore, the base load of a PLC which is induced by the operating system is one parameter for the influence factor. As mentioned in the previous section 'PLC', a profile can be used that includes all the details of a PLC. The operating system and its base load would be a perfect candidate for this set of PLC dependent factors and settings.

Table 4.11: Parameter for the influence factor Operating System

Name	Type	Description
Baseload	ExecutionTime	An exection time as a placeholder for all program and function executions that are induced by the operating system

Scheduling

The scheduling of processes influences the system behavior in many ways. There are several scheduling algorithms or strategies [RS94, GTU91, PS85] available, each tailored to specific requirements and with their unique properties. An influential technique in scheduling operation system processes is preemption. Preemption allows the scheduler to temporarily interrupt a process without requiring its cooperation to execute a process with higher priority. This is a very important property of real-time operating systems. They guarantee that certain processes can be executed without the need to wait for other, lower priority processes. This also reduces the jitter of the process, which is very important for the synchronous execution of motion control tasks [PS01]. The scheduling jitter in real-time operating systems is a deviation from ideal timing event – the delay between the time when task shall be started, and the time the task is actually being started.

Cyclic tasks are configured to run at specific time intervals and execute their associated Programs and Function Blocks within these intervals. If the jitter is too high, these intervals may be reached and even exceeded, leading to a watchdog alarm stopping the PLC and the whole production system. This is an obvious example of the impact of jitter and scheduling. A smaller but none the less important example is the internal processing of incoming data from the fieldbusses. This data must be analyzed and forwarded to the correct memory areas of the tasks. Depending on the firmware architecture, jitter in these tasks may lead to inconsistent data that is used for the calculation of actuator commands. Therefore it is the goal of many real-time operating systems and their schedulers to minimize the jitter and latencies [BFV08].

The scheduling influences the utilization of the PLC indirectly. If for example, all Programs are executed once but the ordering or possible interruptions of other Programs is constant, the overall sum of execution times remains the same. The utilization only changes, if certain Programs are executed more often due to the selected scheduling algorithm. Additionally, in most cases have developers no influence on the scheduling strategy and its configuration. Details to the following scheduling strategies can be found in [Leu04], [PS85], or [RS94].

- First Come First Serve (FCFS) Scheduling: Jobs are executed in a FIFO order, meaning first come, first serve. High priority tasks must wait for low priority tasks that even might wait for a time-consuming IO access. The FCFS is easy to understand and implement, but the average wait time is high, leading to a very poor average performance.
- Shortest-Job-First (SJF) Scheduling: The scheduler knows which processes are ready to run and orders them by their execution time. It is the best approach to minimize waiting time, but impossible to implement due to the lack of information. The scheduler needs to know in advance how much time a process will take.
- Priority Scheduling: As one of the early scheduling strategies, each process was assigned a priority. The process with the highest priority will be executed first. Processes having the same priority level are executed FIFO. The priority can be changed during the execution and may be influenced by the memory, time, or other resource requirements.
- Round Robin (RR) Scheduling: The scheduler defines fixed time interval in which a process can be executed, called quantum. If the process has not finished in its quantum, the process is preempted and the next one continues. There are different approaches regarding the case when a process finishes before the end of its quantum. Some scheduler start right away with the next process, others wait and do nothing. An advantage of this approach is, that it enables a better runtime analysis and reduces jitter in the execution.
- Multilevel Queue Scheduling: This strategy consists of multiple queues, each with its own scheduling algorithms. The queues have a predefined priority. Multilevel queues are often used to group processes based on properties like process type, CPU time, IO access, or memory size. User processes might run with a lower priority than system processes.

- Fixed-Priority Preemptive Scheduling (FPPS): This scheduling strategy is often used in real-time operating systems. A fixed priority preemptive scheduling allows the processor to execute the highest priority processes currently ready. A problem of this strategy is the possible starvation of low priority processes.

The different scheduling strategies each have various configuration parameter or attributes that need to be considered. These could be for example the starvation boost value, the quantum size, length of the time slice, load balancing options, queuing configuration or context switching time. They all influence the scheduling and are entangled with each other. More details on schedulers in (real-time) operating systems can be found in [SGGS98, RS94].

Scheduling strategies can impact the behavior of a PLC and its surrounding automation system. Lower priority process may be preempted and as a result according response times rise. However, they usually impact the overall utilization only slightly. An example is a Task that executes a Program in set intervals. If the Program is preempted too often, the maximum execution time is reached and a Watchdog triggered which stops the PLC. Therefore, to analyze an automation system and the timely execution of Tasks and their Programs, it is necessary to consider at least priorities. It is important to note, that scheduling also covers the dynamic assignment of processes onto cores at runtime. If all processes can be run in parallel without waiting for shared resources, the scheduling can greatly enhance performance [GTU91]. However, setting process affinities is part of the in-depth details of the scheduler and not explicitly covered in this influence factor.

For this reason, yields the influence factor Scheduling only one parameter identifying the used scheduling algorithm. It could be necessary for underlying analysis tools to have this information. However, as mentioned is this parameter usually tied to the PLC and can not be modified separately.

Table 4.12: Parameter for the influence factor Scheduler

Name	Type	Description
Scheduling Strategy	String	Identifier used to set the used scheduling strategy.

File system

Accessing the file system as a shared resource for all tasks and programs running on the PLC has an impact on the program execution. Programs must wait for their turn to write or read data. But during the execution of a typical PLC program like a PID controller, there is usually no access to the file system. All data is held in memory and is written only in small amounts/sizes. This is of course after the systems startup in which all data is loaded from an SDCard or other storage. However, the startup phase is not considered in this case. The only exception from this case is the logging of IO data for analysis. This puts a high load on the file system of a PLC and can lead to a high CPU stress. Due to this reason, the collection/logging of data is often turned on just for a couple of seconds or minutes – and not during normal operation but just for testing. Other services like FTPServer do access the file system to write and read files.

Because the use of data logging is the exception and most PLCs programs are held in memory, the influence of the file system access can be neglected. Load put on the file system caused by additional services like the FTPServer are handled separately in Section 4.6. In case it is necessary to model file access on the PLC, Table 4.13 list the parameters for this low impact influence factor.

Table 4.13: Parameter for the influence factor file system

Name	Type	Description
File size	Bytes	No differentiation between reading and writing. Just the file size is taken into account.
Access	ArrivalPattern	Periodic or aperiodic access on files with a specified size

4.4.3 Firmware and IEC Runtime

Similar to the operating system (Section 4.4.2) the firmware provides and the runtime access to the resources of the PLC and additional functionality.

The firmware consists of management tasks and drivers. The first are used to enable remote debugging, download of new firmware, reading and writing of variables, user and permission management, redundancy, data consistency checks and much more - all depending on the PLC and its vendor. Additionally, there exist some firmware related service programs like Webserver, FTPserver, or OPC-UA Server. They take a very important role in the performance prediction of a PLC, due to their extensive use and specific utilization profiles. These kinds of influences are discussed in Section 4.6.

Drivers enable the access to the hardware and are usually fine-tuned to provide the best performance with respect to jitter and data throughput. Their overhead is minimal and scales with the amount of information that is processed or with the number of calls from a higher level program (e.g. Operating system or IEC Program). Other kinds of drivers are used to control system lights like the power or bus failure Led.

The IEC Runtime is used to execute IEC 61131-3 Code and provide the necessary infrastructure for this task. This includes the interpretation of IEC code (if not compiled into machine code), debugging, hot code replacement during operation, variable monitoring and so on. The IEC 61131-3 [Int13a] standard also defines how variables are addressed and accessed between Resources, Programs and Function Blocks - this must be implemented by the runtime environment.

On top of these standard features, many PLC vendors provide a huge range of extended features and services. For example, Beckhoff industry PCs allow the developers to access the runtime, all its data, and devices via an API called Automation Device Specification [Bec16a]. Other vendors like CodeSys [Smaa] or Phoenix Contact eCLR/ProConOS [Pho16] also provide similar services to access and manage the complex runtimes and to allow software developers to integrate PLCs into non-industrial infrastructures.

Like the operating system, the IEC Runtime (no matter which) and the firmware influence the performance of the PLC. However, the workload that is induced by them – without the execution of IEC Programs, Functions or Function Blocks – is comparable to the background noise of the operating system. All the application related utilization is already covered in Section 4.3. To model them in detail, it would be necessary to use special tools (e.g. profiler) to create fine-grained traces, allowing to map a specific utilizations to process and tasks. For the developer of automation systems, this is not a feasible task. Therefore, these models need to be provided by the different vendors. The Table 4.14 lists only one parameter identified for this influence factor.

Table 4.14: Parameter for the influence factor IEC Runtime

Name	Type	Description
Base load	ExecutionTime	The time it takes for the CPU to execute all background functions of the IEC runtime

4.4.4 IPTraffic

Another influence factor that is directly dependent on the PLCs hardware and the running operating system is the utilization induced by sending and receiving network packets. Each packet that is received at the network interface must be inspected and forwarded to either the IP-stack of the operating system or the fieldbus stack in case the fieldbus is Ethernet based. It is, therefore, necessary to consider the processing of IP traffic. Inspection and forwarding are time- and therefore performance consuming steps, but highly dependent on the hardware, firmware, and operating system.

The most influential parameters are the payload size and how many packets are received at the network interface in a given period. Malformed or damaged packets increase the processing overhead. This is can be the case in industrial environments, where electrical disturbances influence the communication media or if a hacker tries to bring the PLC to its processing limits (denial of service attack) by sending huge numbers of (malformed) packets [LWH05, ACS09]. The goal for each vendor is to optimize the stack [CS00, HJ03] or make it more robust.

There are already several studies that investigate how to model, predict, and analyze IP (internet) traffic and its parameters (see [PKC96, Mah97, FGHW99] or [KLL03] (sec 4.3)). These parameters can often be found in network simulators like OMNeT++ [V^+01, VH08] or NS2 [IH08]. They support modeling of characteristics like error rate, throughput (Hz), or Capacity (Tx Rate). In [HPV15], a MARTE profile for modeling complex networks is provided. However, these models and approaches go deep into the details of IP networks and, therefore, require in-depth modeling and/or analysis capabilities.

For the purpose of estimating a network load in an industrial network and the ease of use, these details should be abstracted to a more common and easy to model factor. The PLC influence factor IPTraffic only considers incoming traffic and not topology, nodes, and accessing devices. Most of the services that will be detailed in the following sections also already include the network traffic that is induced by their provided services. For a further, detailed analysis other tools might be used to investigate each aspect of occurring IPTraffic in the network (see section fieldbus 4.5.2).

Therefore to model the IPTraffic, the first parameter is the number of incoming packets. Both can be specified with an integer value. The size and numer of packets may vary, therefore, stochastic functions like the PMF should be used to set the value. Table 4.15 lists the parameters which are sufficient to model a coarse IOTraffic influence factor.

Table 4.15: Parameter for the influence factor IPTraffic

Name	Type	Description
Number of packets	Integer	The number of packets per time unit
Size of packets	Integer	The average size of the packets

4.5 IO - Fieldbus communication

Inputs and outputs transported and provided by an IO-System are a crucial part of automation systems. Sensors provide input data for a PLC and actuators influence their environment. The data send from and to the PLC is transmitted via fieldbusses. A fieldbus is an industrial network system for real-time distributed control. A PLC might be connected to one or more different fieldbus systems like PROFINET [Pro16a] or INTERBUS [Pro16b]. Depending on the PLC, the communication over fieldbusses is either realized in hardware (e.g. Digital Processing Module (DPM) or Field-Programmable Gate Array (FPGA)), in software only, or a combination of both of them. A communication over a hardware solution also uses a certain amount of CPU time but is significantly faster than a software solution. Due to the real-time constraints, they consume a significant part of the CPU performance. More details on fieldbusses can be found in Chapter Foundations 2.

4.5.1 Fieldbus Examples

There is a huge number of fieldbus technologies available for a wide range of tasks from process or industrial automation, building automation, substation automation, automatic meter reading and vehicle automation applications [Tho05, LaÎ99, Mah13, Ros08, MR04]. An excerpt of these fieldbusses is briefly introduced in the following list. This list does not contain PROFINET, which will be described in more detail afterward. The list is intended to give an overview of the common and non-common features and characteristics of fieldbusses with a focus on Ethernet-based ones. It neither provides a complete list of all their attributes and details, nor a performance analysis of each fieldbus [Pry08, LL02, FFMT04, LMT99, HW00, MDFF06a, PN09].

- **ASI** The Actuator Sensor Interface (AS-i) is a fieldbus usually used for the lower field level process data exchange and not intended for all areas of automation. ASI has been developed during the 1980s by a group of several companies and published in 1994. Today, the specification is managed by AS-International [AS-17] and based on the standards IEC 62026-2 and EN 50295. Two perks of this fieldbus are the (compared to other fieldbusses) lower costs and the possibility to connect all IOs with two cables that can also provide power. The ASI fieldbus requires a master to control up to 62 slaves. The master polls the slaves (usually an IO) with a maximum bus cycle time of 10ms, reaching a datarate of 167 kBit/s. For each slave 4 bit digital input and 3 bit digital output data is addressable. The topology of an ASI fieldbus may be either Bus, Ring, Tree, or Star at up to a segment length of 100 meters – if not extended by additional devices. More information on the fieldbus can be found on the AS-Interface User Organization's [AS-17] website.
- **CANopen** CANopen is a multi layered (industrial automation) protocol that is using the CAN bus for data exchange on the lower protocol levels. Both, the basic communication mechanisms (communication profile), as well as the functionality of the communicating devices (device profile) are defined. CANopen supports up to 127 logical devices, one with master functionality. The underlying CAN bus defines a line (bus) topology. A cycle can be completed within 1 ms on a CANbus running at 1 Mbit, if the conditions are set up correctly (like high priority communication). Each slave can also access the data provided by another slave. The CANopen specifications can be found on the homepage of the Can in Automation (CiA) e.V. [CAN17].
- **CC-Link IE** The CC-Linke IE fieldbus has been developed by Mitsubishi and promoted by the CC-Link Partner Association [CLP17] since the year 2000. Features of this fielbus system are a high noise immunity, a floating master function for higher availability, hot swap of stations (masters and slaves) and many more. It is available as a fieldbus or as a control network variant. CC-Link Field can send 16Bytes of in 2.8 ms (in high speed mode) in a network with 120 nodes.
- **EtherCAT** Developed by Beckhoff, EtherCAT is one of the most common

and fast fieldbus systems available and stands for Ethernet for Control Automation Technology. It supports a flexible wiring and configuration and EtherCAT Slave Controllers can be implemented as FPGA, ASIC, or with standard µController. The master does not need any dedicated hardware - a standard ethernet controller is sufficient. The protocol allows a precise synchronization of the slaves below 1 μs by exact adjustment of the distributed clocks. The tree topology supports cable lengths up to 100 m and (theoretically) 65535 nodes in one seqment. More information about EtherCAT can be found on the EtherCAT [Eth17a] website.

- **Ethernet/IP** EtherNet/IP is an industrial Ethernet network that is one of the leading industrial ethernet networks in the United States. It uses the User Datagram Protocol (UDP) to send IO data. It also allows uploading and downloading of parameters, setpoints, and programs. The protocol uses a producer/consumer system that makes it very efficient for slave-to-slave communication. More information can be found on the ODVA website [ODV17] (formerly Open DeviceNet Vendors Association, Inc.). It supports a star, hierarchical and bus (device level ring (DLR)) topology by using industrial Ethernet switches to 'route' the packages into different collusion domains.
- **FlexRay** This deterministic, fault-tolerant, and high-speed network communications protocol [Par12, Fle10] is the sucessor of CAN and LIN (Local Interconnect Network) with a clear focus on the automotive area. It is an atypical bus system for the automation industry [SJ08] (with little applications) and has been added to this list to indicate that in some cases even domain unfamiliar bus system can be used for automation purposes. The bus is governed by the FlexRay Consortium. It's topology can be set up in a bus, star, or even a mixture of both with a maximum payload of 254 bytes per frame. The duration of a cycle is defined during the design of the network, but is usually between 1 ms and 5 ms. It uses TDMA (Time division multiple access) to build slots for each node in the bus system. The transfer rate can be set to 2.5 Mbit/s, 5 Mbit/s, or 10 Mbit/s. This fieldbus is very atypical for the automation industry, but still provides similar properties for the configuration and use and – despite its automotive focus – in view for industrial applications [SJ08]
- **INTERBUS** This fieldbus [BM94] has been developed by Phoenix Contact and managed by the INTERBUS Club [Pro16b]. It is based on a ring topology in which a master exchanges IO data between multiple distributed I/O modules that are connected to sensors and actuators. Devices in the ring can open further segments, extending the ring. INTERBUS supports up to 512 devices (slaves) with a maximum of 4096 IO points, a transmission rate of 500 kbps, and a max bus length of 400 m. The cycle time required to exchange all IO data depends on the number of devices and data send. The cycle time has a linear growth depending on IO points. As Ethernet-based fieldbusses are becoming more popular, the INTERBUS Club has been integrated into the PROFIBUS International.
- **Modbus/TCP** MODBUS [Mod04] is an application-layer messaging protocol which provides client/server communication between devices con-

nected on different types of buses or networks. Modbus/TCP is the variant which uses TCP/IP networks to send the messages. It focuses on message exchange for non-real-time applications and is widely used. The Modbus server polls each client, which can also implement a server stack as well. For each response and request cycle, the message passes 4 stacks (server → client → runtime → client →server) as well as all switches and routers in between. Modbus is managed by the Modbus Organization, a group of independent users and suppliers of automation devices [Mod17a]. Due to its nature, performance and speed comparisons with other fieldbus systems are not reasonable.

- **Powerlink** This fieldbus technology was introduced by the automation company B&R in 2001. It is now an open protocol managed by the Ethernet POWERLINK Standardization Group (EPSG) [Eth17b]. As the name already suggests is Powerlink another Ethernet-based fieldbus. It requires a special master card that handles the slaves and polls each slave in predefined intervals. The network topology is not fixed, allowing developers to choose a fitting one for the current application. Since version 3, additional features like bulk polling have been added.
- **Profibus** Profibus, like PROFINET, is managed by Profibus International (PI) [Pro17] and stands for Process Field Bus. The technology is pushed by SIEMENS. Currently, two variants are in use. PROFIBUS DP (Decentralised Peripherals) is used to operate sensors and actuators via a centralized controller. PROFIBUS PA (Process Automation) is used to monitor measuring equipment via a process control system in (highly critical or safety relevant) process automation applications. Profibus uses a master-slave approach, whereas multiple masters can communicate with a token ring.
- **Sercos III** Sercos III merges the hard real-time aspects of the Sercos interface [Ser17] with Ethernet. This fieldbus supports up to 511 slave devices which can exchange data with a minimum refresh interval of 31.25 µs. It requires a special master card to manage the slave devices, which can be set up in a line or ring topology. The protocol allows the use of the Ethernet medium to send non-fieldbus data. This data is, however, send with a lower priority and a high chance of being heavily fragmented, based on the set cycle time of the ring/line. Depending on the set real-time properties and the number of slaves, the maximum data size is 50 Byte (cycle time 1 ms, 85 slaves with NRT channel)[Ros08].

4.5.2 PROFINET in Detail

The factors identified in this thesis are to a great part influenced by the firmware analysis project conducted with Phoenix Contact. Therefore, the most detailed fieldbus system under investigation is PROFINET. The focus remains on the CPU utilization, not remote communication effects or response times. These effects can be analyzed by dedicated approaches and are not scope of this thesis.

PROFINET is an open Industrial Ethernet standard developed by Siemens and other partners of PROFIBUS/PROFINET International (PI) and is defined in the International fieldbus standard IEC 61158 [Int16b]. It allows the merging of automation and office networks due to the use of the Ethernet standard IEEE 802.3. PROFINET provides two channels as shown in Figure 4.16: A real-time and non real-time channel for different communication purposes. The Standard TCP/IP channel is used for non-time critical tasks like downloading of configurations, device parameters, accessing diagnostics information, and device management. The real-time channel is used for time-critical data, mainly the automation relevant process data that is send cyclic between the PROFINET master and the profinet devices. The real-time channel is also used for sending alarms, critical messages, and communication monitoring. Possible topologies are line, ring, and star structures.

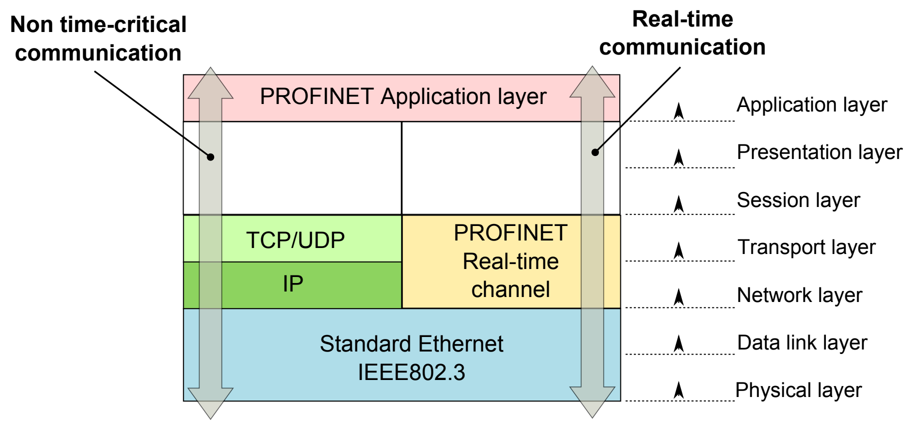


Figure 4.16: The two communication channels and their according ISO OSI layers

The two channels allow PROFINET to support different operating modes ranging from non-real-time (e.g. HMI access) in a component based design named *Class-A*, soft real-time approach named *Class-B* or *RT* for real-time and a class for motion control purposes called Class-C or IRT (isochronous real time). Profibus International (PI) has moved away from the terms RT/IRT and introduced the term PROFINET IO for both RT and IRT. The different classes and their timing requirements ranging from below 1 ms to up to 100 ms are shown in Figure 4.17.

PROFINET makes use of standard protocols for setup, configuration, and maintenance. For example is the Dynamic Host Configuration Protocol (DHCP) used for assigning IP addresses to the different devices. Via the Domain Name Service (DNS) these IP addresses can be resolved by providing easily readable (host)names. For management purposes, the Simple Network Management Protocol (SNMP) can be used to provide information to a broad range of network tools. Lastly, ARP (Address Resolution Protocol) and ICMP (Internet Control Message) are in use for lower level network configuration and management.

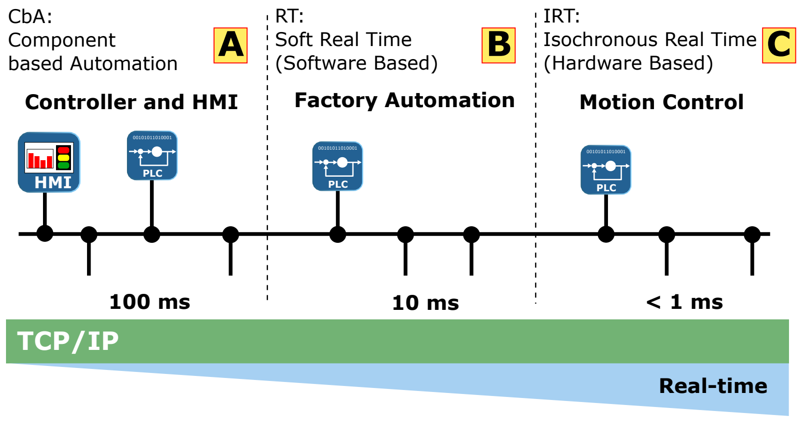


Figure 4.17: Different PROFINET performance classes (modified Figure from [Ros08])

Sensors and actuators producing and consuming inputs and outputs (IO) are connected to PROFINET-Devices (short pndevice) which convert the digital or analog signals into Ethernet frames. These frames are sent in predefined time slots to the PLC and back again. Each pndevice consists of one or more slots, on which sensors and actuators can be connected to. The module size defines the amount of data that can be sent to or from a pndevice. At the PLC, the data is provided to the running programs as variables, also called process data. Therefore, each task running programs with connected process data has a connection to a specific module in the PROFINET. A message sent from the PLC to a pndevice has a fixed length based on the sum of all modules. The devices send interval can be specified independently from the receive interval.

Figure 4.18 shows an exemplary PROFINET device with two modules and three IO points. The pndevice creates a message (PNmsg) with a specific frame structure. This frame is sent in a preconfigured interval to the PLC. At the PLC, the frame is received, analyzed, and the data is copied into the task buffer. The task will take the data upon its execution and copies it again to provide it to the programs. The same procedure in reverse order is used for sending data from the program, over the task to the device. This copying of data is highly optimized due to the fast update times and jitter constraints. Depending on the implementation of the PROFINET stack and the communication with dedicated hardware like an FPGA or a TPS-1 single-chip device interface for PROFINET[2]) the overall PROFINET communication can stress the PLC up to a utilization of 57% for just 16 pndevices [FH12].

There are different network topologies that are supported by PROFINET as shown in Figure 4.19. PROFINET is an Ethernet-based fieldbus and therefore needs additional equipment like switches to relay Ethernet packages between the

[2]The TPS-1 is a highly integrated chip that contains all required components. It supports the conformance class C and can thus be used for all PROFINET communication channels. `https://www.phoenixcontact-software.com/de/profinet-industrial-ethernet/single-chip-interface-tps-1`

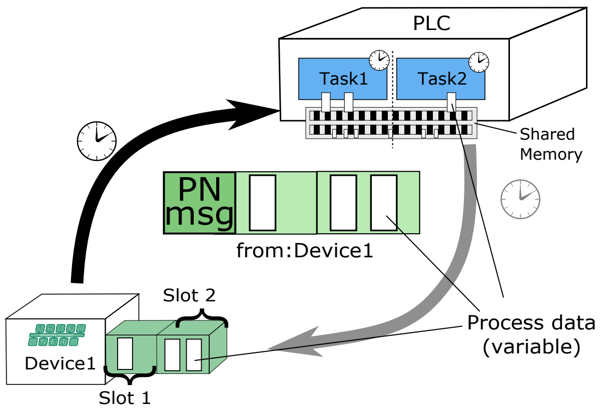


Figure 4.18: Simplified illustration of sending and receiving PROFINET frames

nodes (see [Tan02]). The specific requirements for each device like minimum throughput, performance, jitter, and delays are precisely defined in the PROFINET specification. The Star topology (a) allows the IO-Devices to be grouped in a Star formation around a Switch or any device with switching functionality. Most IO-Devices also include an integrated switch with two to four ports. The second topology is a hierarchical Tree (b). The PROFINET messages must be relayed from switch to switch until they reach their designated IO-Device. The last topology is the Line (c) which is a commonly used and requires integrated switches in each device. So on a physical layer, each PROFINET network can be fitted to the needs of the current automation systems architecture. The underlying functionality CbA, RT, or IRT remains the same, regardless of the chosen topology.

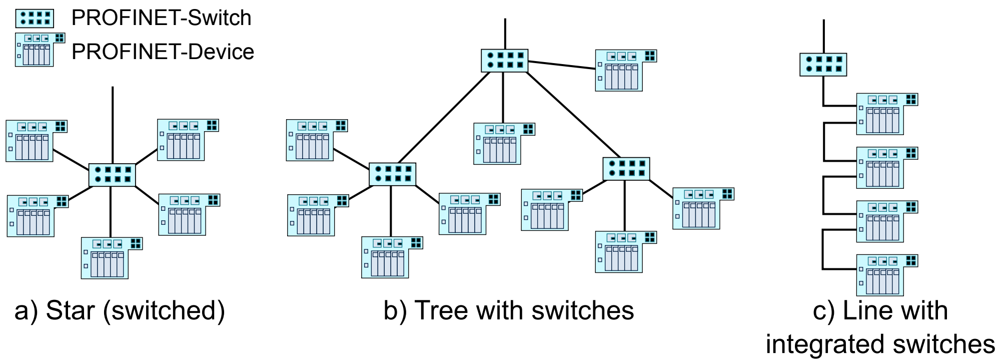


Figure 4.19: Possible PROFINET topologies (mixed forms allowed)

4.5.3 Topologies and Network Devices

Some of the fieldbusses introduced at the beginning of this section allow the selection of a topology, others are fixed to a specific one. The Table 4.16 shows a list containing each fieldbus, possible physical topologies, and the network type.

Table 4.16: Network topologies of the presented fieldbusses

Name	Physical top.	Network type
AS-Interface	Line, Bus, Star, Tree	Master/Slave
CANopen	Line	Multi-Master/Slave
CC-Link IE	Star, Line, Ring or a mix	Master/Slave
EtherCAT	Star, Line or Tree	Master/Slave
Ethernet/IP	Star, Tree or Line	Producer/Consumer
INTERBUS	Ring with branches	Master/Slave
Modbus/TCP	Star, Tree or Line	Client/Server
Powerlink	Line, Bus, Star or Tree	Master/Slave
Profibus	Line, Star, Ring	Multi-Master/Slave
PROFINET	Line, Bus, Star or Tree	Controller/Device
Sercos III	Ring or Line	Master/Slave

The topology has an effect on the performance (cycle time, data throughput, number of devices) of the fieldbus, which will in consequence influence the performance of the PLC the fieldbus is connected to. Especially for Ethernet-based fieldbusses, devices like Hubs, Switches, and Routers impact the overall performance and behavior of the network by adding delays during the forwarding of messages. However, in non-ethernet-based networks, this effect is usually too minimal to be considered in the early systems engineering models. There are several approaches and tools (e.g. OMNeT++, NS2, NS3) that can be used to analyze a network in detail, down to the energy consumption per arriving network packet [Fee01], but they are not in the scope of thesis. The selection of either a star or a tree topology has often only the noticeable effect, that more equipment must be used which slows down the forwarding of messages. However, these effects and additional latencies are usually already considered in the specification of the fieldbus or the requirements for the devices that need to be certified for it. To make a (performance) prediction of even these fine detailed settings, several approaches and tools are already available [SKKS11, LF07, MDFF06b, COH07]. These tools also consider the standard network related parameters like network latency, jitter, forwarding delays, quality of service (priority of packages), package loss, package collisions, and much more.

4.5.4 IO Parameter

There exists a huge number of different fieldbus technologies, each having their own specific attributes and values. To allow the modeling of performance relevant information in the early development phases, the common parameter need to be identified and represented in a fitting, abstract way. The following list is based on the performance evaluations from Phoenix Contact and several other performance modeling approaches. In [LF07] just the PLC, switches, and modules are considered for the simulation of a PROFINET network. TrueTIme [HCÅ03, CHO10] only uses the term network and nodes for a simulation - all other details are hidden in the simulation models and are not accessible for the user. In [MDFF06b], again just switches, IOs, and PLCs are considered in the simulation. They usually neglect the current data size that needs to be transported over a network or the distances between nodes. These parameters are listed here as well to provide a broader information base for subsequent analysis approaches.

Number of devices The number of devices in a fieldbus is parameter applicable for almost all technologies. For PROFINET this is the count of pndevices, for EtherCAT this is the number of slaves. These devices combine several analog or digital inputs into a frame or package and send the data to the PLC. Each IO point that is connected to a device will add to the amount of data send to the PLC. This data needs to be processed, creating a specific utilization on the PLC. Depending on the fieldbus, is the data fixed per device or related to the amount and kinds of IOs. In Ethernet-based fieldbusses, each device also needs to be addressed by one or more network packages. The detailed PROFINET example showed that it is even possible to set varying refresh intervals for sending and receiving per device.

Process data size For fieldbusses the amount of data, which is send over to the PLC and back again, is a parameter that must be considered. Copying data, creating the frames, or parsing the data does impact the PLC and of course the fieldbus throughput. Most fieldbusses use the number of Bytes per package or kbytes/s as a unit of measurement. However, the size is often set by the kind of devices used (e.g. digital or analog devices (see section 2.1.2)). Analog are all devices that provide or use values with more than two states (0 or 1). A light barrier that only sends a signal when an object is detected is an example for a digital sensor. The data this sensor transmits can be represented with just 1 bit. Analog devices can provide more values in a given range. This range is for example depending on the voltage. In a fieldbus, the voltage is converted to a digital signal e.g. 8 bit, ranging from 0 to 255. A temperature sensor may provide a specific voltage for its given temperature range which is then converted to an 8 bit value. Providing a greater data size (e.g. 16 bit) will result in a finer resolution or a greater range of values. However, the classification of a device type as analog or digital is not sufficient for the in-depth analysis of a network (see topology specific model in 5.1.3). The bit size of an analog device may change depending on the fieldbus. The analog/digital property is sufficient for an early analysis. Default values depending on the fieldbus and device type can be assumed for an analog device. For more in-depth analysis, the actual bit size must be provided.

Cylce times The cycle time indicates how often data is exchanged between the

PLC and the devices/slaves. There are usually two ways how the cycle time is determined. The first approach uses an automatically calculated time that is performed offline during the configuration of the fieldbus or in the startup phase. It is usually based on data sizes and number of devices. The second, and uncommon way, is to set it up by the engineer manually (also in offline configuration). Shorter cycles induce a non-neglectable stress on the PLC and must, therefore, be considered in the early design. Some fieldbus technologies are not able to set certain cycles times if the amount of data or the length of the cable/segment is too high. And for some fieldbusses, the cycle time may vary (see ModbusTCP) between intervals. Therefore, a fixed (periodic pattern) value is usually suited for most systems, but a varying value is necessary (including stochastic arrival patterns).

Maximum cable/segment length Interesting - not only for the electrical engineer – is the maximum cable or segment length. This parameter might influence the cycle time of the fieldbus, depending on the used technology. The length of cables and segments is also a requirement that must be considered when choosing a fieldbus for an automation system. Some fieldbusses do not support a cycle time of 1 ms in combination with a specified cable length of 500m. To specify the cable length, the development of the automation system must be in a more advanced stage or estimated. Therefore, for an early analysis of the fieldbus, the length parameter should be taken into account for the creation of the more detailed modeling phases, see topology specific model in section 5.1.3

The different fielbusses have their own properties and requirements that need to be considered. During the analysis of the firmware it was measurable, that the mapping of process data onto the different tasks had an impact on the PLC performance. This effect is, however, highly dependent on the firmware and the protocol stack. Therefore, the mapping of data to Programs and Tasks has not been added to the list of parameters, despite it has been used for previous performance predictions [FH12]. Another factor that might influence the fieldbus behavior is the used cable or medium like fiberglass, copper or other wireless communication like bluetooth [GH09].

To cover all these properties in an abstract model is not feasible. A possible solution for this is the separation into two level of abstraction: A topology independent and a topology specific layer. This approach is further detailed in section 5.1.3.

The Table 4.17 summarizes the important parameters that needs to be considered when using a fieldbus system:

4.6 Services

Services are used to represent the secondary functions of the PLC. Most services provide means for communication with other devices or systems. Well-known

Table 4.17: Influence factor parameter for the io system

Name	Type	Description
Number of devices	Integer	Specifies how many devices are connected to the PLC
Process data size	Integer	Specifies how much data is exchanged
Cycle times	Integer	The refresh interval in which data is exchanged–
Segment length	Integer	Length of the cables/segments

representatives for services, which are available on almost all PLCs, are the FTPserver and the Webserver. Both allow developers to exchange data with the PLC. The following list describes four selected services in more detail. These services are common and cover different aspects of communication, ranging from reading variables over downloading files up to executing operations on a PLC. Most of these services provide their functionality via standardized protocols like HTTP. However, depending on the vendor and PLC, custom services and protocols like Beckhoffs ADS [Bec16a] can be implemented. There are many more services available and their number is – due to the increasing use of software and communication in the automation domain [BJN$^+$06] – rising.

- **FTPServer**: The FTPServer is used to download log files from the PLC, to upload new projects, or to even exchange the whole firmware. Usually, the FTPServer is a low priority process that will get CPU performance when all other controlling tasks are completed. When using the server, only two parameters can be identified: The *filesize* to set the size of the file that is transferred and the access *frequency* to specify how often the server is accessed. Other aspects like the overhead to build up a connection or to authenticate are neglectable in comparison to the data copy and sending utilization. However, there exist some denial of service attacks that just open a connection and wait for a timeout. By doing so, memory is allocated and the server needs more processing time. The Figure 4.20 depicts an FTPServer receiving a project file with a size of 3 MB. It

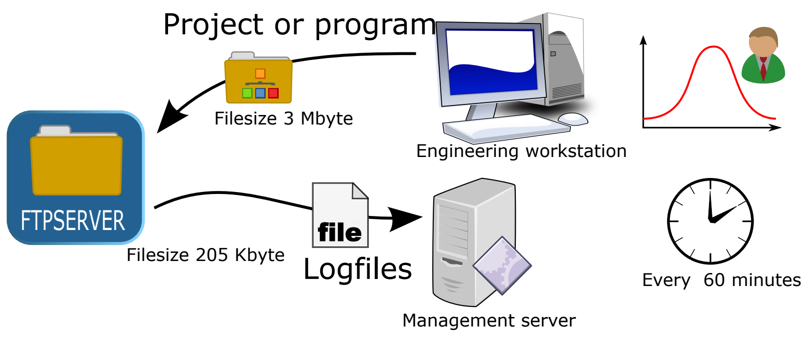


Figure 4.20: Example of different usages of the FTPServer service

is transferred from an engineering workstation every time a bugfix has been created. A reoccurring use of the service is the automatic collection of log files created on the PLC. A management server connects to the FTPServer and downloads one or multiple log files of a given size. This download might happen every 60 minutes, depending on the importance of the log files. How the FPTServer influences the overall utilization of the PLC is depending on various factors, ranging from the implementation of the TCP/IP-Stack, the operation modes (active or passive), or the transfer type (binary or ASCII) of the FTP. For most implementations of an FTPSever, a varying background utilization can be identified. This is the management overhead of the FTPServer which handles the Listener- and Threadpools for possible connection attempts or existing open, but not active connections. The background utilization can also be specified as a *baseload*.

Table 4.18: Influence factor parameter for the service FTPServer

Name	Type	Description
Baseload	ExecutionTime	A background utilization induced by the service for parallel management tasks.
File size	Filesize	Size of the file that is copied.
Access frequency	ArrivalPattern	The frequency how often or in which intervals this service is accessed.

- **Webserver**: The Webserver allows remote users to view the state and variables of the PLC. The variables are copied from the runtime environment and made available for the Webserver process. But this is just the smaller share of the Webserver utilization. Executing server based scripts or executables like CGI and wrapping them into the HTTP protocol takes a lot more time. This generates a certain amount of CPU usage depending on the *access rate* and the *file size* of the website. The number of *accessed variables* must also be considered. Figure 4.21 shows a Webserver being accessed from three different types of devices. First, a Remote Management Server uses the website on the PLC to visualize the current status. In this case, 150 variables must be accessed and shown in a dynamically generated HTML page. The user refreshes the page sporadically. Next, a mobile device like a tablet or phone is used. A typical example is a tablet of a technician inside the factory who is now able to observe the current state of the PLC. Often a virtual private network (VPN) is used to extend a private network over a public network allow the technician to view the state at home from his mobile phone. The last example is a Webserver providing data for a human machine interface (HMI). The HMI is a simple panel able to display HTML pages and mounted at or near the automation system. The HMI usually refreshes the HTML page in a specified interval like 1000ms. Similar to the FTPServer is it possible that the Webserver needs a Threadpool and active listeners to function correctly. Therefore, the Webserver creates a certain amount of *baseload* utilization on the PLC as well.

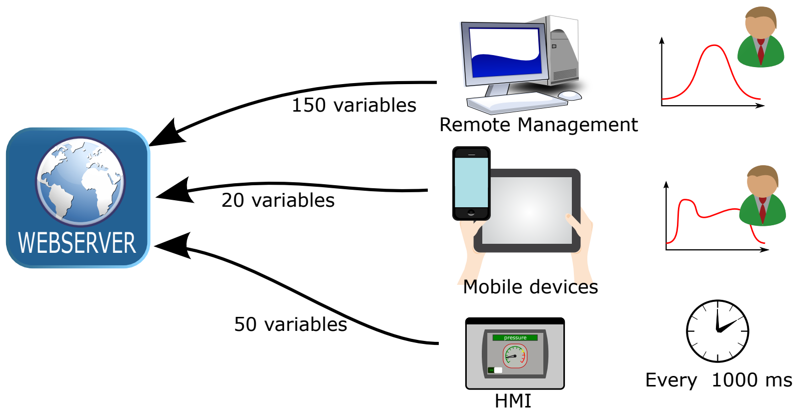


Figure 4.21: Example of different usages of the WebServer service

Table 4.19: Influence factor parameter for the service Webserver

Name	Type	Description
Baseload	ExecutionTime	A background utilization induced by the service for parallel management tasks.
File size	Filesize	Size of the file that is copied.
Access frequency	ArrivalPattern	The frequency how often or in which intervals this service is accessed.
Num. of variables	Integer	Specifies how many variables are copies per access.
Type of variables	Primitive Type	The type of the accessed variable influences the copy process.

- **OPC-UA**: OPC stands for Object Linking and Embedding (OLE) for Process Control and is used to exchange data between control devices. Its successor, the OPC-UA (for Unified Architecture) server, provides services to browse and request variables from the runtime or other PLC related objects. Its main features are the platform independence, integrated secure concepts (encryption, authentication, and auditing), the ability to add new features without affecting existing applications, and a comprehensive information model for defining and retrieving complex information. The OPC-UA server is often used to exchange data between PLCs and other external systems like SCADA, diagnosis or HMI. An OPC-UA client usually requests these variables in predefined *intervals* to update its state. In addition to the rate of access, the *variable type* and the number of variables have an impact on the PLC performance. OPC-UA provides a broad range of other features to interact with the PLC the server is running on. It is, for example, possible to trigger alarms on the PLC that must be confirmed by an maintainer of the automation system. Additionally, is the OPC-Server capable of calling commands which are used to execute functions or applications remotely on the PLC. Figure 4.22 shows some

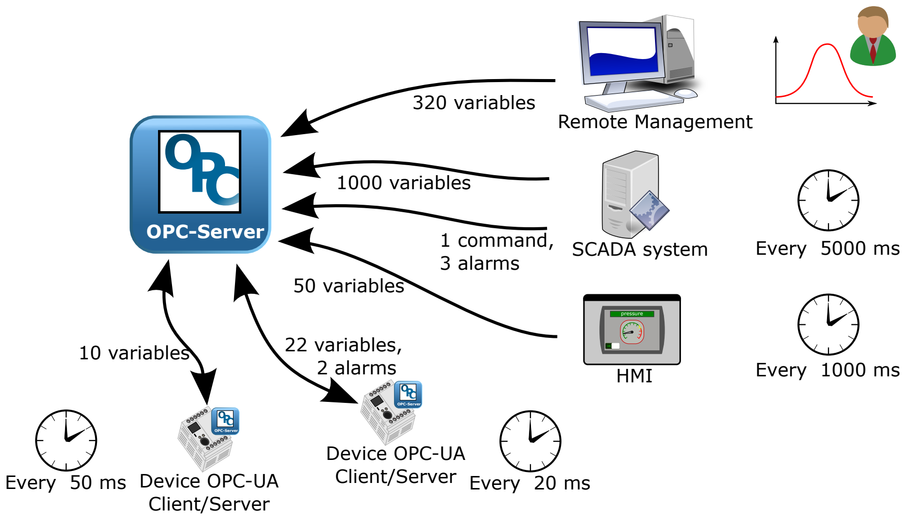


Figure 4.22: The OPC-UA Server and its central role

exemplary scenarios for the use of an OPC-UA server. In the top right is a remote management station depicted that accesses 320 variables to visualize them to a technician. In the second example a SCADA system reads 1000 variables every 500 ms to store them in a database. This information can be aggregated and used for condition monitoring. The SCADA system also reads alarms and is able to trigger a command on the PLC. Next, instead of using a Webserver, the HMI communicates directly with the OPC-Server to visualize the current PLC state. Modern HMI include a small OPC-UA client that is capable of registering on variables to use an efficient publish/subscribe system that sends data only on value changes. However, in this example the data is exchanged based on a fixed interval set to 1000 ms. The last two accesses to the OPC-Server are performed by two devices. One of the goals of the unified architecture is to simplify networking of PLCs from different vendors. This enables the exchange of process data without the use of fieldbusses. For example can alarms and commands be send between devices, allowing a more component oriented structure. This access and communication is not restricted to the field level between automation devices but can also be used to connect the field level with the enterprise infrastructure. The standard even sets the basis for a simple file transfer over the OPC-UA server, making the FTPServer obsolete. The OPC-UA server needs a set of listeners and management threads to function properly, resulting in a *baseload* of this service.

Table 4.20: Influence factor parameter for the service OPC-UA

Name	Type	Description
Baseload	ExecutionTime	A background utilization induced by the service for parallel management tasks.
File size	Filesize	Size of the file that is copied.
Access frequency	ArrivalPattern	The frequency how often or in which intervals this service is accessed.
Operations	Operation	This parameter models the different operations that abstract the behavior of the Function Block.
Load per Operation	ExecutionTime	This parameter must be specified for each Operation. It defines the execution time for each operation call.

- **SNMP**: The Simple Network Management Protocol (SNMP) is commonly used to provide information about the device over the network (IP). SNMP server provide management data as variables structured in an information tree. The structure of this tree is defined in Management Information Bases (MIB). The protocol defines a standard set of variables. Each device, however, can define own MIBs to specificy which kind of information is found in the tree. Management software like WhatsUp Gold [Ips17] can import the device specific MIBs and access these variables to manage and overview the network. In addition to reading variables from the devices, some values can also be set/written over the protocol. Until authentication and authorization was added to SNMP in version 3, this was however a high security risk.

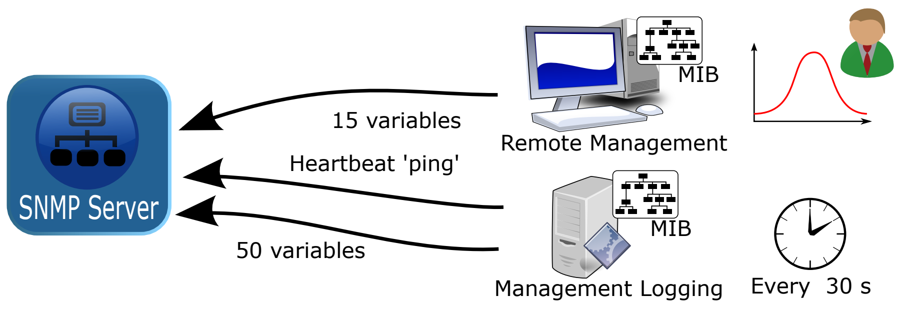


Figure 4.23: SNMP server is providing data for the network management tools

Figure 4.23 shows a device with a running SNMP server. A management station is used to access a couple of variables sporadically. Another server is used to a ping every 30 seconds (heartbeat), checking whether the device is still online. The management station also collectes data about the number of logged in users, current device temperature, or van speed. The stress that is put onto the PLC is again depending on the IP-stack and the operating system. In most well designed networks, the utilization caused by a ping or the SNMP protocol in general can be neglected.

There are many more (vendor specific) services. The list continues with MQTT-Servers [Org17], SQL-Clients [SQL], DDS-Interfaces [Obj04], or Apache Thrift-Interfaces [Apa17]. Each of these services has its unique performance profile when accessing data or functions on the PLC. Common among all services are, however, the baseload that a service uses and the fact that the utilization is depended on the (access) frequency in which an external system or a user interacts with the service.

Table 4.21: Influence factor parameter for the service SNMP

Name	Type	Description
Baseload	ExecutionTime	A background utilization induced by the service for parallel management tasks.
Access frequency	ArrivalPattern	The frequency how often or in which intervals this service is accessed.
Num. of variables	Integer	Specifies how many variables are copies per access.
Type of variables	Primitive Type	The type of the accessed variable influences the copy process.

4.7 Summary

In this chapter, the various influence factors that impact the performance of a PLC in an automation system have been investigated. These factors have been identified based upon common factors from existing performance prediction approaches, by exhibiting the firmware of a well know PLC vendor Phoenix Contact, and by analyzing exemplary System Engineering models of automation systems. This list of influence factors is gathered depending on overall impact on the PLC, how difficult it is to model/capture them, or whether information is available in the early development stages. A short summary is given in the following list.

- **Program:** The execution of Program code directly influences the utilization of the PLC and the automations systems performance
- **Function Block:** Invoking a Function Block and its optional background load puts stress on the PLCs CPU.
- **Cyclic Task:** The CyclicTask is a periodic task with fixed intervals. The task is used to trigger the execution of a Program or Function Block.

- **Event Task:** This aperiodic task is hard to model due to the different sources for events which trigger the execution. An approximation with density functions is best to incorporate their induced utilization.
- **Idle Task:** The idle task is dependent on the execution time of associated Programs and Function Blocks. The interval of the task may jitter.
- **PLC:** The PLC as a container for several hardware related factors like CPU frequency, caching, architecture and multi core influences the execution of Programs and Function Blocks, services, and all other computational factors.
- **OS:** Operating system processes create a base utilization on the PLC. For embedded devices these processes are stripped down to a necessary core. The base utilization can be neglected, but the impact of this factor is low.
- **Firmware & Runtime:** Management processes and Runtime Environment influence the code execution and create a background utilization. Usually the performance oriented design puts low stress on the CPU and is therefore a low impact factor.
- **IPTraffic:** The major part of IPTraffic is covered in the IO and the service influence factors. However, a tight integration with an office network or attacks may (in rare cases) affect the automation system
- **IO:** Receiving, processing and sending of process data is are CPU intensive tasks. Their impact on the performance of a PLC are high. Several parameters need to be considered.
- **Services:** There are various types of services already available and in development for future PLC generations. They become more and more popular, hence their influence on the PLCs performance can not be neglected. When and to what degree the performance of the PLC is influenced is depending on the service and the vendor.

These core influence factors must be considered when conducting an analysis and/or developing an automation system. As stated in the beginning of this Chapter and in the detailed introductions of each influence factor, are most details only important for a precise analysis of the PLC. Even small changes in the compiler settings or task priorities can influence the execution behavior of an idle task. But these effects are usually hard to model or are simply not available in the early stages of the development. Forcing developers to consider each and every little aspect might not lead to the desired early evaluation of the system. Also, by providing too much details, the prediction of the performance will become harder and might lead to wrong results. Starting with more abstract, high-level models to perform performance predictions and then detail these models to conduct in-depth analysis should be preferred [Jai90]. In Chapter 5, these factors will be used to create a formal model that captures the relations between factors and all their parameters.

CHAPTER 5

Automation Influence Model and Development Process

To plan a complex automation system, a team of developers needs reliable information to choose the right components, fieldbus technology, or to find an optimal software deployment. Using Systems Engineering methods, the developer team is able to create a first design model of the system. This model usually focuses on hardware and electrical components. But over the last couple of years, software became an increasingly important part of the overall automation system [Vya13, SN99]. This is forcing developers to consider software artifacts like programs or external services in the early development phases as well. System Engineering methods like CONSENS [GFDK09] allow the modeling of hardware, software, and different kinds of flows to indicate an exchange of information, energy, or material. CONSENS also provide means to model requirements and to relate them to the model elements.

Current System Engineering methods in general support the development of automation systems. However, domain-specific performance relevant information is neglected. It is possible to use performance modeling profiles like MARTE [Obj06] to provide this additional information, but they usually focus on the annotation of software relevant factors only or are domain-unspecific and tailored towards a detailed timing analysis. In Chapter 4, influence factors and their various parameters are identified. To capture the influence factors for a subsequent analysis, a formal model is needed that can be automatically analyzed or transformed into analysis models. To validate the early design of an automation system development, this formal model should be integrated into existing Systems Engineering models. Additionally, a process that guides developers to capture and structure the influence factors for each automation system is needed. This is necessary, because an analysis can only be conducted if a certain level of detail is achieved and the needed influence factors and their relations can be modeled.

Contribution C2:
Method for Modeling Automation System Influence Factors
The second contribution of this thesis is a method for capturing automation specific influence factors in the early phases of the development. This method consists of a formal model to capture the influence factors based on Systems Engineering models and a process to guide developers during the specification.

CONSENS has been selected as the foundational Systems Engineering approach which is extended by model elements to capture the influence factors. CONSENS provides means to model complex systems on a high abstraction level, providing developers of multiple disciplines the possibility to synchronize and coordinate themselves. To capture all performance relevant influence factors, a formal model has been developed that extends CONSENS. This model is called the Automation Influence Model, or short AIM. The second part of this contribution is the definition of an accompanying development process. This process is used to guide System Engineers through the steps of adding influence factors and their parameters to existing or new CONSENS models.

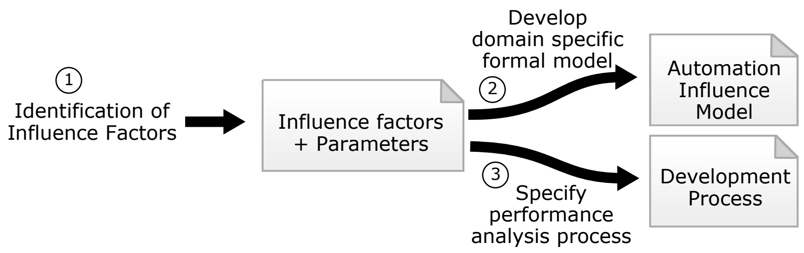

1
Identification of Influence Factors
Influence factors + Parameters
Develop domain specific formal model
2
Automation Influence Model
3
Specify performance analysis process
Development Process

Figure 5.1: Overview of steps and artifacts in Chapter 5

This Chapter is split into two parts. In Section 5.1, the influence factors, their parameters, and their relations are formalized in a class model (Step 2 in Figure 5.1). It it discussed, what information needs to be available at which development step to specify the influence factors or how detailed the Systems Engineering models need to be. This covers the various levels of detail that can or must be achieved, the refinement of components, and the specification of hardware dependent and hardware independent software. The second part, Section 5.2, focuses on the process that guides automation system developers through the specification of influence factors (Step 3 in Figure 5.1).

5.1 Automation Influence Model

The Automation Influence Model (AIM) is used to formalize the different influence factors and how they are related to each other. A formal model provides well-formed syntax and semantics, making it (automatically) process- and analyzable. This is a necessary prerequisite to run automatic performance predictions based on the influence factors an automation system developer provides. The formal model is developed in this chapter and is used as a basis for the UML profile created in Chapter 6 to evaluate the developed method.

To define a formal model, it is necessary to identify a set of elements, their properties, and relations between them. For this, three questions need to be answered first:

- **Required modeling depth:** The modeling depth will affect the development process as well as the number and kind of elements available to developers to create the model. It must be taken into account, whether developers will be able to refine certain elements into subsystems and how this will impact the overall model. In Section 5.1.2 it is discussed, why refinement is necessary, how detailed a model must be, which elements can be refined, and what impact a refinement has on the model.
- **Topology independent models:** The fieldbus influence factor needs considerable effort to model all relevant elements. But in the early stages of the development, the kind of fieldbus is usually not fixed. It is therefore necessary, to enable developers to first model the fieldbus independent of a technology and topology. In Section 5.1.3, a method for refining modeling topology independent into topology dependent model elements is presented. This impacts the overall design of the formal model and its elements.
- **Hardware dependent and independent Load Specifications:** Software with a fixed execution time runs on each PLC for the same duration regardless of the actual hardware (e.g. CPU speed). Therefore, the execution time must be estimated or measured for each PLC, leading to a high modeling effort. To model the execution time independent from the PLCs hardware, additional elements and properties must be made available in the formal model to capture different kinds of loads. Section 5.1.4 introduced these different kinds of load specifications.

Before discussing these questions, related work considering similar problems or formal models for performance modeling in general are presented in Section 5.1.1. Afterward, the specification of the Automation Influence Model is detailed in Section 5.1.5. Class diagrams are used to capture the various elements, their properties, and relations between them.

The Automation Influence Model already abstracts from certain influence factor details like the PLC's CPU caching and focuses only on aspects which are essential to model/annotate in the Systems Engineering models. This improves the overall modeling process reducing time and effort to annotate existing Systems Engineering models. Influence factors that are neglected in this formal model are be provided by the more detailed analysis models (see Chapter 6), which take aspects like multicore, priorities, and the caching into account.

5.1.1 Related Work

Formal models are used to specify quality of service attributes for a range of approaches. Most of the analysis tools provide their own input models and specification language to model the different parts of the automation system they want to analyze. In the following, a short selection of formal models to capture performance specific information is given. A comprehensive list of other formal models can be found in the survey form Balsamo et al. [BdMIS04].

Another survey, with a focus on modeling and predicting the performance of component-based systems, has been conducted by Becker et al. [BGMO06].

MARTE (Modeling and Analysis of Real Time and Embedded systems) [SG13, Obj06, HPV15] is a well known UML Profile for annotating existing UML models. It is a general approach that allows to annotate UML models with performance specific information with a focus on real-time and embedded devices. The stereotypes defined in the profile allow developers to define elements like semaphores, concurrent tasks, or schedulers and scheduling policies. Software elements can be annotated with execution times, resource usages, and relations among each other. To model hardware specific elements, the pool of resources contains for example processors, memory, input and output devices, and networks. The application of UML and MARTE to predict the performance of software systems is also content of the book by Koycheva [Koy13].

In [GM04], a language has been developed for representing performance-related properties of components and of their composition. This language can be processed and a compositional performance analysis conducted. Components are annotated with services they use or provide. Examples for lower level services are a network with properties like bandwidth or bytes_sec or a CPU with speed and number of operations per second. Via connectors, these services are linked and a complex system is composes. Afterward, their component based approach is mapped onto the SPTP [Obj05] UML profile and made available for other analysis approaches.

The Palladio Component Model (PCM)[BKR09, RBB^{+}11] is an architecture description language supporting performance evaluations of component-based software systems. The language and meta models are formally specified with the EMF Framework [Ecl17b]. Palladio enables developers to model high performance software systems that are distributed on different processing resources and connected via networks. The framework and meta model allows developers to create extensions that provide user defined model elements and/or additional analysis. It is for example possible to add different scheduler for processing resources like an operating system which can assign different processes to CPUs or cores. For this, basic scheduling algorithms and a framework to specify custom scheduler (see [Hap08, Hap04, Hap16]) can be used. All meta models and language definitions are described in detail in the Palladio [RBB^{+}11] tech report.

In [PMDB14], the focus is set on the model driven development of safety-critical systems. They define viewpoints and provide a method for the multi-view modeling of hardware platforms. By integrating their viewpoints and the support for hierarchical and variable horizontal composition of hardware platforms into the MechatronicUML [DPP^{+}16] modeling language, they enable developers to generate runnable code for the modeled targets. In their approach they make use of a resource model that allows the specification and parametrization of embedded devices like micro-controllers or control units. This model can be composed of atomic computing resources like processor, memory, flash, or communication resources with, for example, their physical layer attributes like data rates and protocols. These hardware models in combination with software annotated by WCET, allow subsequent performance analysis. MechatronicUML and its extensions are based on the EMF Framework [Ecl17b] to formally specify meta models.

In [FK98], the Quality of Service modeling language (QML) is described which provides means to specify quality of service attributes for interfaces, operations, operation parameters, and operation results. The modeling language allows to model contract types to define the dimensions of quality attributes like performance as delay in milliseconds and throughput in mb/sec or reliability in Mean-Time-To-Failure (MTTF) or numOfFailures. The contacts can be used to understand/specify the quality of service requirements for individual components as well as the complete system under development. Frølund and Koistinen also show how their contract types, contracts, and profiles can integrated with UML to model complex software architectures.

In [AS00], Arief and Speirs present an approach for an UML tool and its models to automatically generate simulation programs. They use UML class diagrams to model the static structure and sequence diagrams to model the dynamic properties of a system under investigation. With their Simulation Modelling Language (SimML) they provide a formal basis to model elements like process, data, or queue. They provide information that can be used by a statistics component. This component collects data that is relevant to the performance evaluation of the simulated system. The annotated model is transformed into Java code and executed to perform a discrete-event process-based simulation. As a result, different quality of service properties like processing delay, average service time, or number of processed jobs can be predicted and the design of the software evaluated.

In 'UML-Based Performance Modeling Framework for Component-Based Distributed Systems' [Kah01] an approach to model performance relevant information in existing UML models. They show how to model distributed software systems considering three basic resources: CPU usage, hard disc access, and network traffic. It is possible to define complex factors by combining the basic resources in combination with an additional queuing or delay behavior. Via the textual Performance Modeling Language (PML), the resource usages can be added to classes and operations. The annotations for accessing CPU and disk for a given operation would be defined as *Write()* {*cpu=10,disk=80*}.

The paper 'UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models' [dMLH$^+$00] by de Miguel et al. introduce a UML extensions to represent temporal requirements and resource usages. These extensions add a set of formal constraints and stereotypes, which can be used to model general latency and capacity quality of service requirements. Part of these constraints and stereotypes are definitions to model periodic timing constraints, execution times of uml elements, stereotypes to define networks and processors, or cyclic classes to define the temporal distribution of cyclic operations. The UML diagrams extended by the annotations are used as input models for the automatic generation of scheduling and simulation models for different tools. Their formal model to specify quality of service attributes and artifacts is based on UML, stereotypes, and OCL[Obj17] constraints.

Conclusion

This section introduced related work and provided insights to selected approaches. Most of the analysis tools provide their own input models and specification language to model the different parts of the system they want to analyze. They focus on a specific goal or target area and do not cover the wide range of influence factors identified in Chapter 4. However, they share similarities, common elements, and techniques to create (automation) system models. These common elements are considered during the definition of the formal Automation Influence Model in Section 5.1.5, as well as meta questions regarding the required modeling depth (see Section 5.1.2) and the need to provide hardware independent models (Section 5.1.4).

5.1.2 Required Modeling Depth

Modeling complex (automation) systems is a time consuming task that needs a high degree of communication and interaction between the different involved disciplines. The (simplified) goal of Systems Engineering approaches is to provide a process and models to create an overview of the whole system including all involved disciplines. As a result, Systems Engineering models are a first, partially rough sketch of the system under development. In most cases, at least some parts of the system under development need to be modeled in more detail to provide enough information for the involved domains or to conduct appropriate analysis. This leads to two points that need to be discussed before deciding which model elements must to be considered or created.

- **Question 1:** When to stop going into further details and start with the discipline specific development.
- **Question 2:** Which level of detail needs to be achieved to be able to specify a subset or all influence factors.

These points are discussed in the following.

Question 1: The first, general problem has been tackled before in various domains, ranging from common modeling [BRS95], over software development [SVC06, SVE07], to performance modeling up to Systems Engineering [Alt12, BRS95, CBB15]. A partial answer can be given by the three characteristics of a (software) model as given by [Sta73, HM08] or [MSUW02]. First, the model needs to abstract properties and remove details of the modeled object. This is the *Reduction Aspect* or *Abstraction* property of a model. It will make the model less detailed as its real world counterpart, resulting in an easier to understand representation, allowing to focus on the important properties only. Second, the real world object with all its considered attributes is projected (Projection Aspect) onto its model. *"A model can be seen as the result of a projection (in a mathematical sense). The real world object is projected onto its model representative by removing the unconsidered attributes. This projection is an isomorphism if the projection of the real world entities on the model entities still allows conclusions to be drawn from the model entity onto the real world entity with respect to the aim of the model. The term refers to the equivalence (iso= equal, morph = shape) between the model and the real world entity."* [Bec08]. And third, a model must have a purpose and is pragmatic. This allows to use the model instead of the real world object to answer specific questions under given assumptions.

Therefore, the question how detailed a Systems Engineering model (of an automation system) needs to be, depends on the use of the model. Usually the focus of Systems Engineering models like CONSENS is to create a holistic overview of the system. The Vee-Model for the CONSENS development process suggests that after the principle solution has been fixed, the different disciplines start their designs based on the coarse structure of the system. However, this rigid process is now replaced by a more iterative approach [Rie14, HBM$^+$15]. But there is still no guideline or process how to find the correct level of detail for a specific system under development.

Question 2: The second problem, knowing which level of detail needs to be achieved to be able to specify a subset or all influence factors, can be easily answered for the majority of influence factors. These factors rely on the PLC as a crucial part of the automation system. This includes – of course the PLC itself – as well as the Tasks, Programs, Function Blocks, Functions, and Services. As a result, to model these elements, it is necessary to provide a containing or referable PLC. Section 5.1.5 details the different model elements that make use of these relations to a PLC.

Function Blocks, Functions, and Tasks can be defined anywhere in the automation system, including the PLC, subsystems, or IOs. This has the advantage that certain software related elements can be a part of their according hardware element, creating automation specific modules. It is necessary to reference the PLC that will execute these software elements.

The services must be a part of the PLC. They are provided by a PLC and can not be defined outside of it. For example, the FTPserver is a component of the PLCs firmware and can not be run or modeled on a different element. Services are preset by the chosen PLC type. Elements that make use of the provided Services are not dependent on an existing PLC, but must reference them for the service access.

For fieldbuses and IOs, the question what level of detail needs to be achieved to model all required elements can not be answered with a simple solution. The general purpose of a fieldbus is to connect the various IOs with the PLC for data exchange purposes. On a high level of abstraction, the type of fieldbus, the

IOs, and the PLC are sufficient to derive information for subsequent analysis. However, for more sophisticated analysis, the fieldbus specific elements like buscouplers, connections, or nodes must be considered. The following example illustrates how a fieldbus can be specified on different modeling depths.

Figure 5.2 shows an excerpt of the *Unit2* of the turbocharger production system. The full example of this unit is presented in Chapter 3. The FT-Robot is communicating with the Mill to know when to release the clamps and the gripper. This is a more functional view that shows how both components need to interact with each other. The only modeled elements are the components and their exchanged information. However, the communication between these elements must be exchanged via a fieldbus and is usually handled by a PLC. Such a communication impacts the performance of an automation system, respectively the PLC. Omitting the fieldbus and its properties will lead to an imprecise prediction. Therefore, this level of detail is not sufficient.

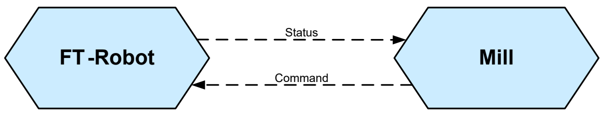


Figure 5.2: Active structure with a functional view on communication

The fieldbus and its incoming and outgoing flows are added in Figure 5.3. Neither the FT-Robot, nor the Mill have computational units themselves. They are just subcomponents coordinated via the PLC as the heart of the automation system. The PLC collects data from both subsystems and also runs/executes the Programs and Function Blocks controlling the behavior of the system. The logical view has been detailed by a more technical view on the system, introducing a processing resource (PLC) and a fieldbus. The *Fieldbus* element represents the fieldbus that is used to relay all exchanged data. This information is crucial for the development, because now the electrical engineers as well as the software engineers know, which component is executing the programs and that a data exchange between the PLC and the subcomponents must be realized via a given fieldbus. This level of depth is sufficient to model the fieldbus, its properties, and connected IOs or elements. There are alternatives to model the fieldbus. One of them is to use type each information flow, connector, and port with a specific fieldbus. But these elements would not be shown directly in the diagram, making the model more complicated and harder to understand.

To incorporate all fieldbus specific influence factors and parameters, the level of detail must be increased on more step. The electrical engineer and the mechanical engineers need to know, how the information is exchanged. This means, that details about the used technology, necessary wiring, additional (communication) components, or topology are needed. Figure 5.4 shows a more detailed view including two Buscouplers and an Energysupply. The sensor/actuator data from the Mill and the FT-Robot is not send directly to the PLC, but is relayed by a fieldbus specific bus coupler component. These buscouplers provide

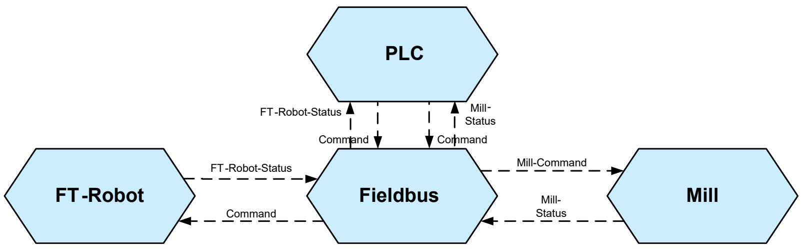


Figure 5.3: Active structure with a more technical view on communication

information to conduct an in-depth performance analysis of the fieldbus. Depending on the fieldbus type, the number and properties of the fieldbus specific elements can vary. See Section 5.1.3 for a detailed analysis of fieldbus specific elements.

The Figure 5.4 also contains the energy supply needed by the Buscoupler, PLC, FT-Robot, and the Mill. The energy flows are used to provide electrical energy to the Energysupply unit. Usually, in this modeling depth each component is not directly connected via a energy or information flow. For this, Ports are used which type the information, signal, or energy and enable a much more formal and precise modeling. However, the ports are omitted in this figure due to the emphasis on the buscouplers which are actually used to connect the different system elements.

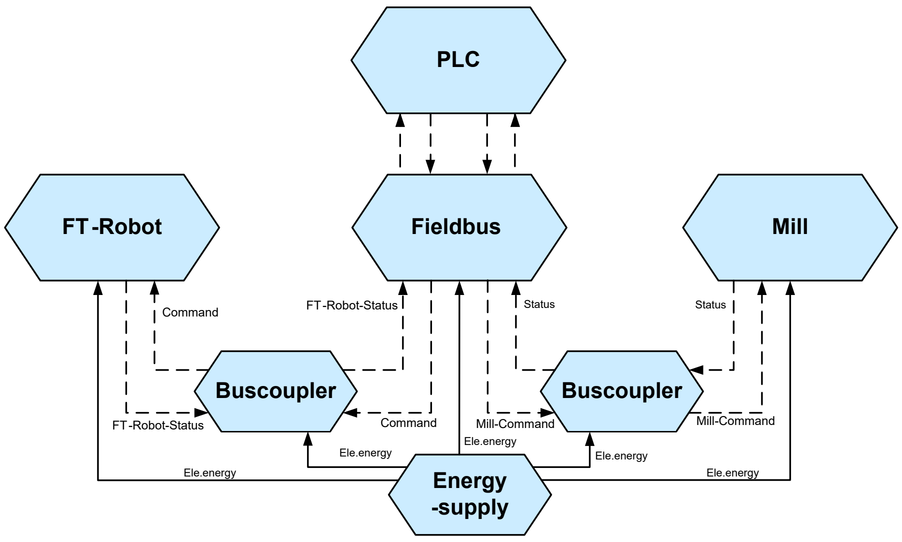


Figure 5.4: Active structure with a detailed model of the communication

Conclusion

The two questions answered in this section showed that the depth and detail in which a model is created highly depends on the purpose of the model. To incorporate all influence factors in the Systems Engineering models, at least basic information needs to be available. This includes the PLC, Tasks, Programs, Services, and IOs. These elements must be added in order to perform a performance prediction. Regarding the fieldbus, two different levels can be identified: an abstract level only considering basic information and IOs and a more detailed view covering the topology of the fieldbus. These two levels of detail are further detailed in Section 5.1.3.

An important point that needs to be considered is the refinement of elements into smaller parts. Programs and Function Blocks are software based examples for such refinements. A Function Block can be composed of further Function Blocks. The Automation Influence Model must be capable to handle such refinements.

Another example of a hardware centric refinement is given in Figure 5.5. It depicts how the hardware modules *AssemblyConnector* element and *PLC* are refined into smaller components. On the higher level of detail, a developer estimated that the *AssemblyConnector* will consist of nine IOs and a total datasize of 132 Byte. In the following development steps, the *AssemblyConnector* will be further detailed, allowing the specification of each single IO with its own data size and kind. In the figure, the information flows connecting each IO are combined into a single flow to provide a better visualization.

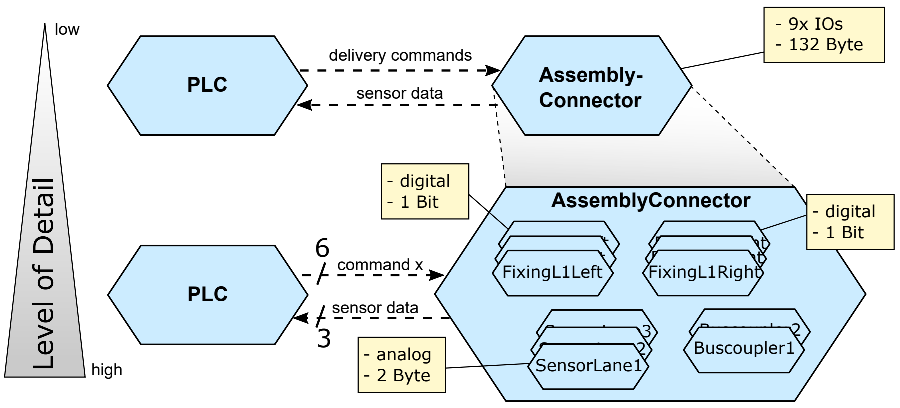


Figure 5.5: Refinement of the AssemblyConnector containing several IOs

As a result, the Automation Influence Model must be capable to model high-level elements that optionally can be refined in later stages of the development. For this, the topology dependent and topology specific level have been specified, which are presented in the following section.

5.1.3 Topology Independent and Topology Specific Model

As identified in the previous section can some influence factors be specified at different levels of detail. This also applies to a fieldbus or network used to provide a communication channel between the PLC and IOs. In these early stages, a highly detailed specification of the fieldbus and all its components and settings is usually not addressed. The bus systems and networks are roughly sketched and later refined.

However, in case an early validation of the system design should be conducted, all fieldbus specific information must be added to the model to fully describe the influence factor. This would force developers to provide each buscoupler, switch, hub, fieldbus setting, and IO configuration already in these early stages. As a result, the model would incorporate information that might not be needed at this point and therefore violate the model reduction aspect. Additionally, changing the type of fieldbus in the early stages could result in an in-depth and time consuming restructuring of the model.

To counter this problem, two consecutive models are provided: the topology independent model and the topology specific model. The first model provides elements to specify a fieldbus with very few elements on a high level of abstraction. It is sufficient to conduct a basic performance analysis and provides the foundations for a subsequent refinement into the second, more detailed model. This second model includes all, fieldbus-specific details that must be incorporated into the Systems Engineering models. This approach will allow developers to focus on the important elements of the model and reduce time and effort to create or update detailed fieldbusses.

Topology Independent Model

The Topology Independent Model (TIM) provides all basis influence factors including PLC, Tasks, Programs, Function Blocks, Functions, and so on. It also covers an abstract fieldbus and IO definition, that can be used to roughly sketch the communication parts of the system under development. However, it does not include topology specific model elements like network nodes, buscouplers, connections, or network interfaces. The fieldbus can reference all necessary elements without the need to specify how they are connected or witch topology is used.

When conducting a performance prediction only based on the TIM, various assumptions must be made by the analysis or preceding transformation steps. Basically all minimum required parameters must be set either randomly, by a default value, or derived from the current model.

Taking PROFINET as an exemplary fieldbus specified with the TIM, it will contain the fieldbus type (PROFINET), the desired or minimal interval time and several IOs. For the analysis, the IOs must be assigned to currently neglected buscouplers. A simple way to do this is to just assume each buscoupler can manage eight IOs, no matter if they are digital or analog. Also, each buscoupler will be set to a given refresh interval and data size to exchange. When conducting a performance prediction based on the TIM, the developers must keep in mind that serveral assumptions are made and that the results of the prediction can deviate from the measurements and from the more detailed

Topology Specific Model (TSM). However, this way an early performance prediction covering the major influence factors of the system is possible, without the tedious task to model all fieldbus specific elements and parameters.

Topology Specific Model

The Topology Specific Model (TSM) is used when the kind of fieldbus has been fixed and a level of detail is achieved that allows the modeling of fieldbus specific elements like buscouplers, gateways, bridges, routers, switches, and so on. The TSM provides more detailed IOs properties, allowing the developer to specify parameters and settings tailored for each device. However, changing back from the detailed TSM to the more abstract TIM might result in in-depth corrections, spanning throughout the model. Fieldbus specific elements must be removed or given properties values updated. Figure 5.6 depicts, how the TSM model extends the elements of the TIM providing fieldbus specific influence factors and parameters. The TSM makes use of the high level elements specified by the TIM and either references or extends them to provide fieldbus-specific elements. A major part of this extension is based on the IO elements and properties defined in the TIM.

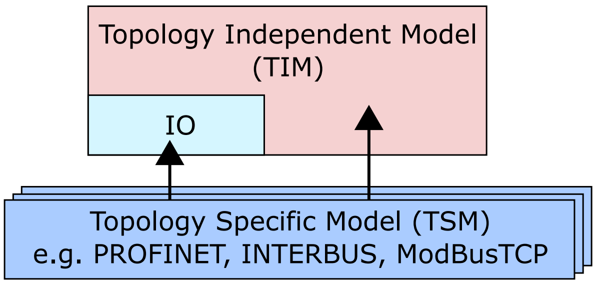

Topology Independent Model (TIM)
IO
Topology Specific Model (TSM)
e.g. PROFINET, INTERBUS, ModBusTCP

Figure 5.6: Topology specific parts extending the TIM base

There are two ways the TSM can be designed. Either as a holistic model containing all network devices and settings which can be generally applied or as multiple extensions for each kind of communication technology available for an analysis. Having to include all current and future fieldbusses, covering the broad range of network devices - without just defining an abstract network-node element -, and incorporating all of their specific properties would be an impossible task. The TSM would steadily grow and cover a wide range of fieldbusses. For this thesis, the second design choice to use multiple, fieldbus-specific TSMs has been selected. This approach is more flexible with regard to upcoming fieldbusses, their unique model elements and parameters, as well as necessary additions to the analysis.

Conclusion

Using these two kinds of models, developers are able to specify influence factors at varying levels of detail. This allows a rough analysis of the system without the need to model each topology dependent element in the early stages of the development. In the TIM, changes to the underlying fieldbus will result only in minor model changes like setting the interval time, name, or version of the fieldbus. The TIM can be refined until a mature state is reached or the decision for a fieldbus has been made. From this point on, the TSM can be used to set up the fieldbus and topology dependent model elements and their parameters. Splitting up the TSM into different, fieldbus specific parts further pushes the extensibility and flexibility due to future changes and additions of fieldbusses. This approach also matches with the proposed realization by using UML profiles for each TSM (see Chapter 6).

5.1.4 Hardware Dependent and Independent Load Specifications

By simply using execution times (e.g. $50ms$, $3sec$) to define how much a Program or Function Block will stress the PLCs CPU is a valid strategy. Execution times can be estimated and provide developers a natural way to set the time it takes for a software to execute and how much load it will put on the PLC. However, using fixed execution times limit flexibility, extensibility, and portability. A Function Block with a fixed execution time runs on each PLC in the same duration, but should be finished on faster CPUs much earlier. Therefore, the execution time of the Function Block must be estimated or measured for each PLC, leading to a much higher modeling effort.

A solution to this problem is an abstract demand of a software. A resource like a CPU can process a certain amount of this demand in a given time. This allows to model the software independently of the hardware. In the following, three different ways to model load specifications are presented. A fixed time as an *ExecutionTimeSpecification*, an abstract demand via a *ResourceUsageSpecification*, and a *ModelUsageSpecification* that indicates that the load is covered by the analysis model and not defined via the influence factor. They each offer different advantages when modeling specific influence factors.

- **ExecutionTimeSpecification:** The basic and most intuitive approach is to use a fixed time value. This value must be provided by the developers of an automation system and is usually estimated or measured from existing software. The value specifies a time how long an execution will take on a given CPU. There are various ways to specify such a time value, ranging from Worst-Case-Execution-Times to Probability-Mass-Functions (see Chapter 4). The Wort-Case-Execution-Time, for example, allows a developer to estimate the time a Program will run in the worst case scenario (= the longest duration possible). Specifying such a time value will also function as a requirement for the Program, forcing the developer to uphold the given time boundaries when programming the software. Setting a fixed time value for the execution of a program is an option available in most performance modeling approaches like MARTE [SG13, Obj06], TrueTime [HCÅ03], OmNet++ [Ope17], or the Timing-Architects Tool Suite [TA17].

- **ResourceUsageSpecification:** By setting a fixed execution time, the actual performance of a PLC is not considered. The set execution time can be estimated or reused from previous projects, but as soon as the (performance class of the) PLC changes, the execution times will likely not be applicable for the new system. A faster PLC could execute a program in less time, on a slower PLC in more time. Therefore, it is necessary to provide another option to specify, how much load will be put on a CPU running a Program or Service execution.

  Depending on the analysis tool and its features, a literal is provided that is parsed during the analysis. In this thesis, the Palladio Component Framework is used to run performance simulations to predict the PLC utilization in an automation system. Palladio uses resources and specifies demand with SEFF (see 2.3.2) to calculate the utilization of a resource. An exemplary resource is the CPU which can process 15000 resource units per time unit. Specifying a usage of 5000 per time unit will result in a utilization of 33%. MARTE provides the Value Specification Language (VSL) to set resource usages based on execution times. With a expression like $exectime = ((117 * \$CPUspeed), us)$, a hardware independent execution time is set. By using a variable *CPU speed* in a formula, it is later possible to reuse the element without changing the model or creating a different model for every hardware setup.

  The Timing-Architects Tool Suite allows the specification of *Runnables* that are executed on the hardware. Each runable can contain several instructions to perform, representing different algorithms or code[1]. The hardware is also modeled separately with the capability to execute a certain amount of instructions per time unit. TrueTime ([HCÅ03, CHL+03, COH07]) allows the specification of functions. These functions can contain runnable code or simulated execution times specified by delays.

  Using an analysis tool specific to model resource usages will make the simulation much more accurate. However, finding the correct resource specifications and identifying the resource usage is often quite complicated [Jai90]. More information on how to model resource specific usages is given in Chapter 2 and an example evaluation including the modeling of resources and resource usages shown in Chapter 6.

- **ModelUsageSpecification:** In the last option, the developer can neither specify the execution time nor resource usage. The details are hidden in the analysis models and are PLC specific. This is, for example, the case when selecting and parameterizing a fieldbus. The utilization for

[1]AMALTHEA Tool Platform Help - `http://www.amalthea-project.org`

this influence factor is calculated based on given parameters and not on an execution time a developer must specify in the Systems Engineering model. Special hardware like FPGA or ASIC might boost the performance when using a specific fieldbus on a selected PLC.

Another example are PLC specific service operations. The access of services and their operation parameters can be modeled, but not the actual stress they will put on the CPU. They too, rely heavily on vendor, PLC, or even firmware specific implementations that might boost the execution of certain services.

Conclusion

Three alternatives for modeling the effect of an influence factor have been discussed. The first one is the hardware dependent definition of specific **execution times**, the second one is the definition of hardware independent (abstract) **resource usages**, and the third one is the usage of non-modifiable, PLC specific features which can only be parameterized. To capture these alternatives, Figure 5.7 shows the different kinds which are grouped under the term *LoadSpecification*. The hardware independent branch contains the *ExecutionTimeSpecification* which specifies a fixed time with the WCET, BoundedExecution, RandomSetExecution and RandomIntervalExecution as specified in Section 4.2.4. The hardware dependent branch provides a specification for a resource based usage or with the indirect way over the model parameters. And the last leaf on the right side specifies the *ModelUsageSpecification* to define that the analysis model is used to specify the load that is put on the PLC.

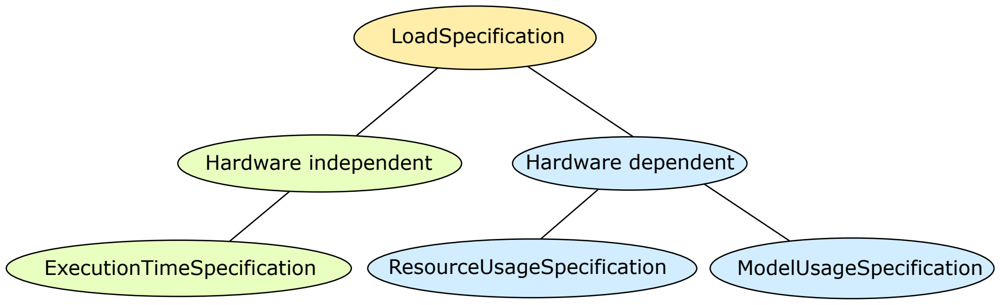


Figure 5.7: LoadSpecification and different sub types

Each of these alternatives can not be applied to every influence factor. The following Table 5.1.4 lists the influence factors and their applicable *LoadSpecification* types. A short, supplementary description provides further details. Please note. that the terms *ExecutionTime*, *ResourceUsage*, and *ModelUsage* are abbreviations for the previously specified *LoadSpecification* subtypes to better fit the table.

Table 5.1: Influence factors and applicable LoadSpecification type

Factor(s)	**Usage**	**Description**
Program	ExecutionTime, ResourceUsage	Either a fixed time value or a resource usage can be used. A Program is not a part of a PLC and therefore no program should be specified by the model.
Function Block	ExecutionTime, ResourceUsage, ModelUsage	Each alternative can be used. Basis Function Blocks can be provided by the PLC/vendor (e.g. IP, MySQL,..), reused from previous projects or estimated.
Function	ExecutionTime, ResourceUsage, ModelUsage	Function provide the same alternatives. They can be part of the PLC (e.g. TON, ADD, ...), reused from previous projects or estimated.
CyclicTask, EventTask, IdleTask	ModelUsage	Only model parameters are used. The parametrization of a task will result in a CPU load. Vendor and PLC specific.
CPU	ModelUsage	Part of the analysis model, resp. the PLC. Can not be modified by the developer.
OS	ModelUsage	Part of the analysis model, resp. the PLC. Can not be modified by the developer.
Firmware & Runtime	ModelUsage	Elemental part of the PLC that must be modeled with in-depth knowledge of the PLC. Can not be modified by the developer.
File System	ModelUsage	Similar to CPU, OS, and Firmware is this part of the model and should not be modified by a developer.
IPTraffic	ModelUsage	The parameters of incoming IP traffic can be given, not modeled in detail how much load will be put on the CPU.
IO	ModelUsage	A coarse and detailed IO model can be specified that resembles the parameters for this influence factor. How much stress will be put on the CPU is highly dependent on the PLC and its supplementary hardware (e.g. a FPGA).
Services	ModelUsage, ResourceUsage, ExecutionTime	Services are usually provided by the PLC and therefore primarily modeled as parameters. There are however some services that might be provided through additional software that are in the scope of the developers responsibility. Therefore the CPU utilization of this factor can be modeled with all three alternatives.

All remaining influence factors like compilers, caching, or additional hardware are part of the underlying analysis model and are therefore of kind ModelUsage.

5.1.5 Model Specification

The Automation Influence Model consists of multiple parts that are detailed in the following subsections. Each part, namely PLC hardware, POUs, Tasks, IOs, and Services, covers a group of influence factors and their parameters. The

IO part will be specified in more detail by providing the previously introduced Topology Independent and Specific Models.

PLC

Most of the performance specific details of the PLC like the firmware, operating system, scheduling, or runtime must be covered by the detailed analysis models. Therefore, only a few parameters are visible/modifiable to a developer of an automation system. Figure 5.8 shows the PLC, its attributes, and containment relations. To identify a concrete PLC and use the according, detailed analysis models the *articleNumber* is used. In most cases, this number is not human readable, therefore the *PLCIdentificationID* provides a more easier to read identification string. The *vendor* attribute can be derived from the articleNumber and is an additional property to increase human readability.

The *IPTraffic* is used to model the incoming IP packets in a very simplified way. The number of packets and the size can be specified - both with a fixed, bounded, random interval or random set modifier. Another way to model the ip traffic could be profiles that include certain patterns which occur in a factory. Examples for such patterns are a timed collection of log files or the deployment of regular updates onto multiple PLCs. These patterns are omitted in this thesis and could be part of future work for network specific TSM. Function Blocks, Functions and Programs are contained by the PLC. They are triggered by Tasks. More information on these model elements can be found in the following sections.

In Figure 5.8 two more enumerations are depicted. The *ExecutionTime* enumeration provides a set of literals to specify which kind of execution time (WCET, Bounded, RandomInterval, or RandomSet) is used. The execution time of, for example a Program, could also be specified with a string literal. However, providing an additional enumeration value simplifies the parsing of the string literal and constrains on the model. The second enumeration is the AccessPattern. It is created for a similar purpose, but with the different literals Periodic, Bounded, RandomSet, and RandomInterval to specify arrival patterns.

POU

To specify Programs, Function Blocks and Functions, two approaches can be followed. The first approach is more focused on CONSENS. The elements in the active structure do not follow a type and instance system. Environment-, System-, and SolutionElements can be compared to prototypes. This is a convenient way to create systems engineering models easily, fast, and also fits the mindset of mechanical and electrical engineers best. Following this approach, each POU would be defined directly in the corresponding usage specification.

The other alternative to model the POUs is a strict type and instance system. This forces the systems engineers to specify Program, Function Block, and

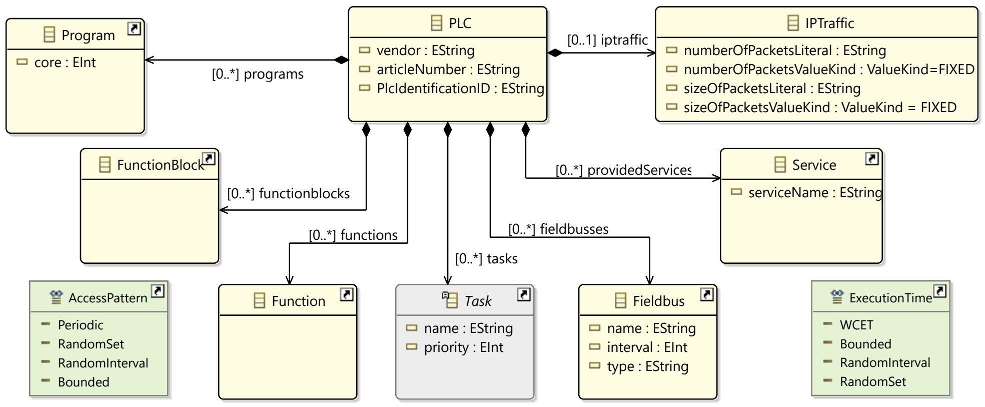


Figure 5.8: Formal model of the PLC, its attributes and containment relations

Function types in a first and to use these types in instances in a second step. This improves the reuse of POUs within a complex Systems Engineering model as well as the reuse of specific elements from previously created automation systems. However, reusing POUs specified with a fixed execution time may lead to incorrect performance predictions in case, the POU is executed on a faster or slower resource (CPU). Therefore, it should always be validated if the specified values of the selected POUs are still applicable for the current model. This problem will not occur if, instead of a fixed execution time, a hardware independent resource usage has been modeled (see section 5.1.4).

For this thesis, the selected approach for specifying POUs is based on the well known type and instance system. Using types and instances improves the usability and is already known by IEC 61131-3 developers. Additionally, the profile that is presented in Chapter 6, is based on SysML4CONSENS which also uses a strict type and instance system (Blocks and parts). Therefore, the types for Programs, Function Blocks, and Functions must first be specified as containments of the PLC.

This close relation to the PLC has the advantage, that it is easier to indicate what Functions and Function Blocks are specific to a certain PLC and/or vendor (e.g. Phoenix Contact specific Function Blocks for database access). An alternative would be the introduction of a POU library container. Such a library, embedded in the system model, could also be organized by vendors and/or PLCs.

Figure 5.9 shows an excerpt of the influence model. Each PLC contains a set of *POU* types. The *POU* is an abstract class that is refined by the Program, Function, and Function Block. The Program is able to use Function Block instances (*FunctionBlockInstance*) and call Functions (*FunctionCall*). It has just one property *Core* which allows to set the affinity of the Program. Function Blocks and Functions are executed every time its parent Program or Function Block is executed. The Program itself is triggered by a Task. *Functions* can be called via *FunctionCalls* and *FBOperation* are called via *FBOperationCalls*.

For each Function or Operation Parameters can be specified (see *FunctionParameter*, *FunctionCallParameterSpec*, *FBOperationParameter*, and *FBOperationParameterSpec*). Parameter can further be detailed with the *type* property. Its value for a call is set via the *parameterLiteral* in its according ParameterSpec.

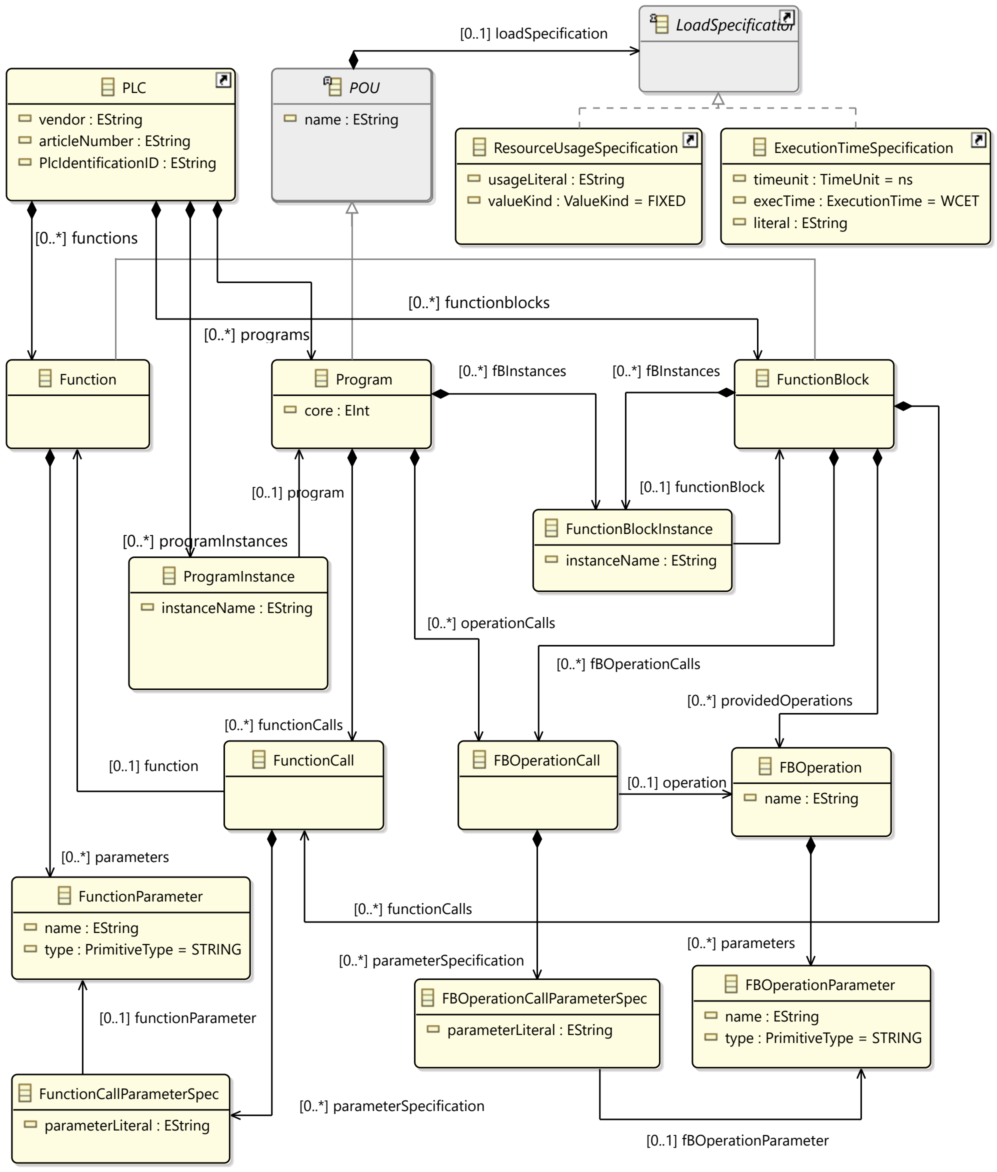


Figure 5.9: Classes overview for Programs, Function Blocks, Functions, and related elements

To model the influence on the PLCs performance, the abstract class *POU* provides three alternatives by adding or omitting a *LoadSpecification*. Omitting a LoadSpecification will lead to a model usage of this POU. This means, that the underlying analysis must provide the load this POU will put on the PLC. In case an element *ExecutionTimeSpecification* is provided, the specified attributes *execTime* and *timeunit* can be used to estimate the execution time of the POU. This execution time is, however, used on all PLCs this POU has been instantiated on, regardless of the execution speed of the PLC. In case the *ResourceUsageSpecification* is provided, the POU uses an abstract resource demand that is specified for a specific analysis tool. In this thesis, the Palladio Component Framework is used to perform a simulation and determine the utilization of the PLC. Therefore, the POU needs to be specified with a PLC independent resource demand. More details can be found in Chapter 7.

Tasks

In Chapter 4, three different types of tasks have been identified and their parameters specified. The Figure 5.10 shows the relations between the PLC, Tasks, and Programs. Each PLC contains a set of tasks which can be either of type *CyclicTask*, *EventTask*, or *IdleTask*. Each task executes one or many *ProgramInstances*, which are typed by a *Program*. The CyclicTask provides an attribute to specify the *cycleTime* in which the task is periodically executed. The *EventTask* is further parameterized by an *AccessPatternSpecification* to provide information when the task should execute the associated programs. The *IdleTask* inherits the priority from the abstract *Task*, but does not provide further properties to specify. The behavior of the IdleTask is vendor and PLC specific and, therefore, must be covered by the tool that will perform the analysis.

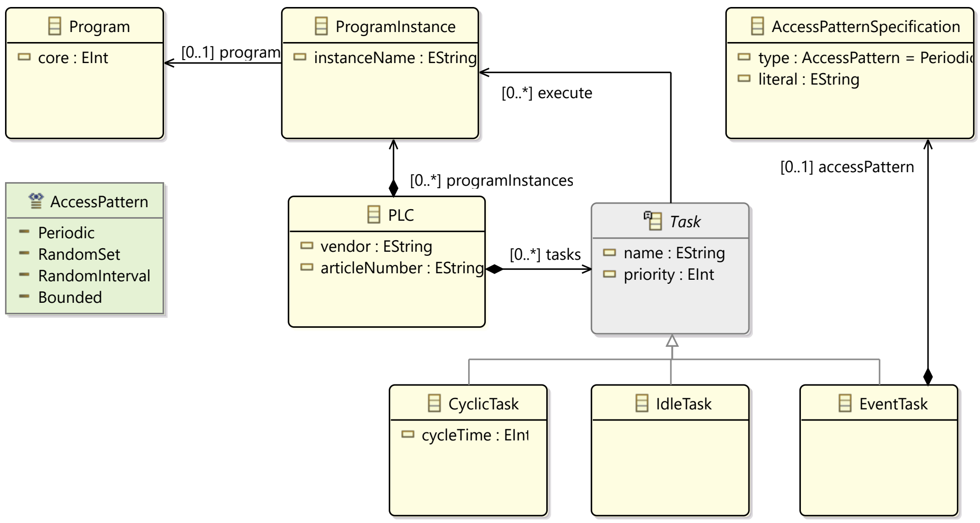


Figure 5.10: Classes overview for Tasks and related elements

The separation into different subclasses instead of a single Task with an access pattern (e.g. the *Task* class), allows in the “Analyze System Performance” step of the proposed development process (see 5.2.2) an easier transformation into input models for the performance analysis. In this step, it is necessary to distinguish between an IdleTask and an EventTask due to the different scheduling algorithms that might be used – depending on the vendor and PLC (see EmbOS Scheduler in Chapter 6).

Services

In this section, the classes and relations to model services and the access to these services are introduced. There are several ways to do this.

The first approach is to model each service as an independent factor with its own, specific parameters. An influence factor for the FTPServer would include the properties file size and the frequency/distribution of accesses from the user or other external systems. Another dedicated OPC-UA influence factor would be configured with a list of send variables, their sizes as well as a similar frequency/distribution for accessing the data. It would be necessary to add a new influence factor for each new service. The advantage of this approach is that every influence factor can be matched to exactly one service and the parameters can be used intuitively and fit the factor perfectly. Figure 5.11 shows a couple of services, each specified as an influence factors with their dedicated parameters. This approach has also some other benefits. It would be easily possible to

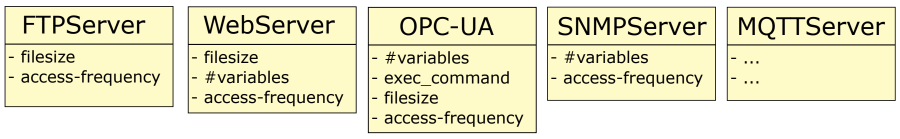


Figure 5.11: Services as influence factors with individual parameters

find suitable PLCs just by defining the necessary services. A list of PLCs from different vendors could be automatically filtered to constrain the selection of a fitting PLC.

The second approach would be to identify the common attributes of all services and group them together in one, abstract influence factor which provides a broad range of attributes. In the near future, flexible and autonomous production plants force the exchange of data on various levels throughout the production process [Ver12, VDI, Kag12, GB12]. This will lead to a strengthening of existing communication standards as well as to the development of new services. Therefore a single, abstract service influence factor is a more flexible approach to specify and model new services and their specific parameters. A similar approach has been used for the specification of SysML stereotypes [Obj15b]. Several stereotypes provide a rich set of properties which can not be applied to an element at the same time. More information on SysML can be found in Chapter 2. Figure 5.12 shows a graphical representation of the influence factor *Service* which can be used to model different kinds of parameters. The parameters are as follows. The number of variables that will be accessed is a property that can be used for the OPC-UA Server or Webserver. It is therefore a general property that should be considered. The access frequency has already been identified as a common parameter for all services. Several

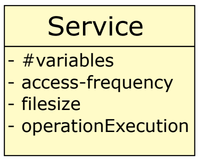


Figure 5.12: The (abstract) *Service* influence factor

services allow to request files. The OPC-UA server, FTPServer, and even the Webserver can transmit data. In the listed examples, only the OPC-UA server is able to execute operations. However, other services like Beckhoffs ADS support [Bec16a] calling functions like the *ADSRDSTATE* block. Future services will also be able to execute or trigger operations and programs on the PLC. Therefore this property must be taken into account.

In case, a dedicated influence factor is used for each service, the specifics of this service must be represented by this factor. Due to the increasing number of services for automation systems, it is not feasible to update the list of factors when necessary. Additionally, using just one service would require all involved developers of an automation system to synchronize with each other and successively add parameters like variables and operation accesses to the defined service.

Therefore, it should be possible that each developer (in its discipline) can model the parameters independently. During the analysis, all parameters must be merged and assigned to the appropriate service. Due to the fact, that the analysis tools/models must be updated/configured for each existing and new service anyways, this would be an ideal way to reduce the overall modeling effort.

Figure 5.13 shows an example of modeling each service usage of an OPC UA server independently. The parameters shown are specified by one or more engineers and tailored for specific tasks in the automation system. The access of 50 variables is set up by the electrical engineer who needs process data for the execution of a smaller device. The transfer of a log file to a management server every 5000ms is modeled by an software engineer. The third factor on the left side models the use of a command send to the OPC-UA server. The *updateCache* function is executed in an interval of 5000ms. On the right side a smaller device is accessing 10 variables every 10ms. The last factor shows an access with a distribution function. The *RandomSetPattern* defines an access every 12 minutes with a probability of 5%, every 15 minutes with 90% and every 17 minutes with 5%.

Basically each access is modeled and the parameters are transfered to a corresponding service in the analysis. Therefore, the services are not explicitly modeled with the possible features they provide, but rather are the sum of all modeled accesses. In chapter 6, the UML profile used to annotate the Systems Engineering models is specified. The profile allows to define such services. The analysis is able to identify these services and use the corresponding analysis

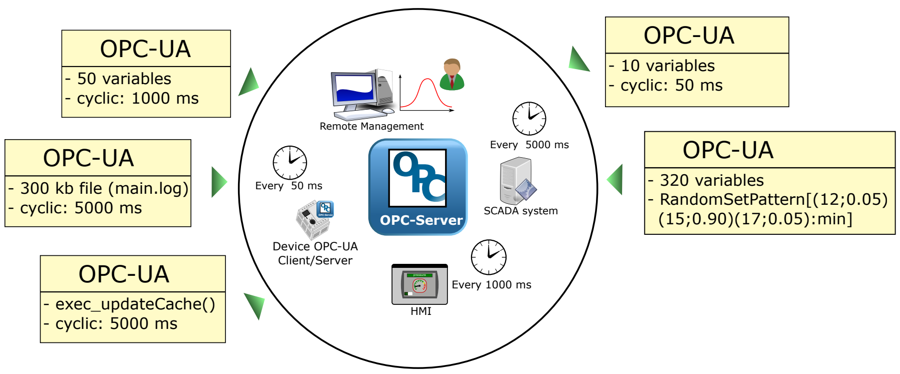


Figure 5.13: Combination of multiple influence factor specifications into one

models and load profiles. All accesses that set the parameters for the factor are transferred to the analysis model and provide the necessary input for the performance prediction. This allows to model accesses independently of the service and with a focus on a task, rather then the implementing server. PLC vendors can provide their own profile extensions and according analysis models to incorporate new or modified services.

The according classes, properties, and relations to model the influence factor service are described as follows. Each *PLC* may contain zero or more *Services*. Usually the number and kind of services provided by a PLC is depending on its type. Different vendors and performance classes of a PLC might offer varying sets of services (e.g. without an OPC-UA server or just with a simple webserver). While the number and kind of services might be fixed by the PLC, the developer/engineer often has the ability to disable a service which will not be used. In this thesis, the developer is responsible for setting up the provided services. This also includes the specification of *ServiceOperations* and their *ServiceOperationParameters*.

Elements which can use the services must be of kind *AbstractElement*. For the application in CONSENS, *SystemElements* or *EnvironmentElements* would, therefore, be subtypes of AbstractElement. Three different kinds of *ServiceAccess* are considered in this thesis. The *VariableAccess* represents a query on a set of variables. Its parameters are the number and type of variables accessed. The *FileAccess* only needs to specify the size of the file which will be transfered from the service provider to the requesting element. The third type is the *OperationAcces* which is used to model an execution of a service operation. A Service needs to provide operations which the *OperationServiceAccess* must reference. Each *ServiceOperation* may provide zero or more parameters which can also be referenced by the according *OperationParameter*.

As mentioned are some services PLC or vendor specific. Selected PLCs may actually not provide a service or operation that has been modeled. Therefore, before running an analysis, appropriate checks must be made to find inconsistencies between the influence model and an analysis model. This check is not covered in this thesis.

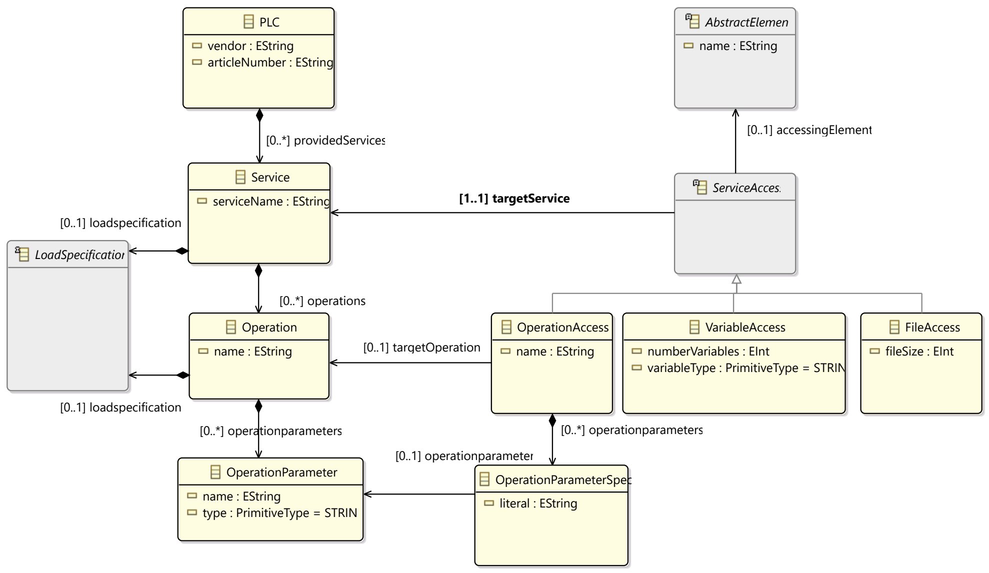


Figure 5.14: Classes overview for services and related elements

IO

One crucial problem when modeling the influence factor IO system, is the large number of parameters that need to be taken into account. For a PLC, the compiler, caching, scheduling, and many more parameters may affect the actual performance. These “fine tuning” parameters are usually included in the detailed models of an specific analysis tool for each PLC. The same applies to the modeling of fieldbusses. There are several general attributes that are shared among all fieldbusses (see influence factors in Chapter 4) and other important features such as the topology or the type and performance of devices used in a fieldbus.

As mentioned in section 5.1.2 is it necessary to support multiple levels of abstraction. To model the influence factor IO and its parameters this thesis uses the two levels *Topology Independend Model* (TIM) and *Topology Specific Model* (TSM) to separate fieldbus independent information from fieldbus dependent. The TIM includes the common elements and attributes that can be applied on any fieldbus (see Chapter 4). On the TSM level, the fieldbus specific information must be specified. This includes for example cable lengths, segments, collusion domains, topologies, and other network related properties. The two modeling levels are described in more detail in the following subsections.

Abstract models for the fieldbus (TIM) The main task of a fieldbus system is to connect the sensors and actuators to the PLC so that data can be exchanged between them. The data from the IOs is usually converted to variables that can be accessed by the POUs running on the PLC. A common property for each fieldbus is that all data provided by the sensor and actuator has a specific size. The cycle in which the data is updated depends on the POU execution and/or on the interval in which the fieldbus gathers data from each IO. Figure 5.15 shows the classes to model the influence factor IO on this high abstraction level.

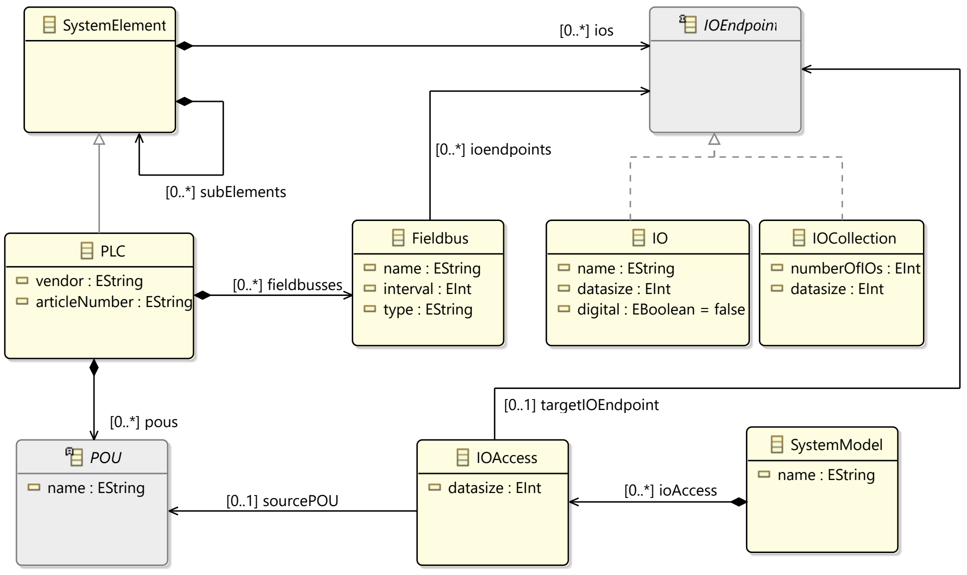


Figure 5.15: Classes for modeling an abstract fieldbus

Each PLC may provide none or multiple fieldbusses, depending on the type of PLC and its configuration. The *Fieldbus* provides three properties: a name for the identification, a type specification and an interval in which the IOs will be refreshed/polled. The type is used to later select a fitting simulation model and to generate the necessary input models and their modeling depth. Each fieldbus contains zero or more *IOEndpoints* which can either be an *IO* or a whole *IOCollection*. The *IO* is a simple actuator or sensor which provides information about the *datasize* it will provide or need. An additional property *digital* is used to indicated whether the IO is digital or analog. The IOCollection is a convenience class which can be used to model a complete set of IOs and estimate their aggregated *datasize*.

To model the access of, for example a Program to a sensor, the *IOAccess* can be used. This relation class connects a POU with an *IOEndpoint*. It provides an additional attribute to specify the amount of data that is accessed. This attribute is only necessary if the *IOEndpoint* is an *IOCollection* and just a subset of the IOs is accessed.

Depending on the approach and tool, the abstract model does not provide sufficient information to conduct a (performance) prediction. Details about the underlying media, topology, refresh times, and many more must be available to create valid models for an analysis. Therefore, it is necessary to derive or assume additional model elements and parameters. A simplified example is shown in

Figure 5.16. The abstract TIM provides just the IOs and, to some extend, also details about the kind of IO (e.g. digital or analog). For a simulation, a transition has to be made which adds a fixed cycle time – depending on the number of IOs – and specifies the underlying topology. The IOs are evenly distributed to added fieldbus specific buscouplers. These necessary transitions can be made semi- or fully automatically, depending on the used fieldbus. More sophisticated simulation tools with different analysis aspects like OmNet++ or TrueTime, may require other or additional data to conduct a sufficient analysis.

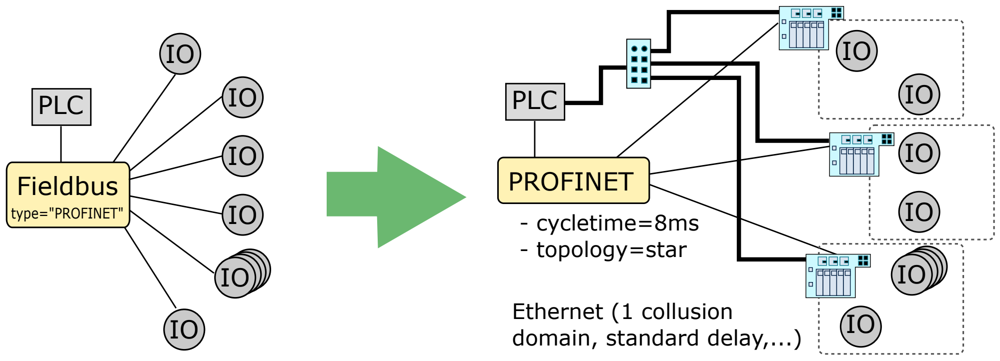


Figure 5.16: Exemplary PROFINET transition from TIM to TSM

Detailed models for the fieldbus (TSM) The detailed model (TSM) provides the developers the ability to model the fieldbus in much more depth, once the fieldbus has been selected for the system under development. For this, the abstract elements of the IO can be extended by different, topology specific models to provide these details. Examples for such details could be new fieldbus specific devices like buscouplers, switches, and router or common attributes like cable/segment lengths, jitter, delays. These details provide more information for the influence factor IO and extend the set of parameters.

Due to the sheer number of different fieldbusses and their various attributes, it is not feasible to include them all in one model. Therefore, additional models providing information about a specific fieldbus are needed. (Abstract) classes can be extended by these models, allowing the developer to capture more detailed information about network topologies, devices, further attributes, and settings. However, in case these fieldbus specific elements are added to a system model that was previously very abstract, the system model will lose its fieldbus and topology independent nature. A switch to another fieldbus in the later phases of the development will therefore cause extensive change effort in the system model.

The following examples present two different topology specific models necessary to detail the PROFINET and INTERBUS fieldbus. For an analysis, it is often necessary to have sufficient information about all modeled influence factors. In case a fieldbus has been selected and all its specific components and properties are modeled, this information is available and can be used as an input for e.g. a performance simulation. The presented models do not cover each element and property in detail. They are merely used to indicate, how a more detailed model can be created and how an integration into the TIM is possible. Note that there are exists various ways to model the structure of a fieldbus. The TSM below should give an idea how these models can be build and focus on an easier mapping onto the CONSENS UML profile in Chapter 6.

PROFINET TSM: The core elements of this package contain all information that is necessary to model a more detailed view of the PROFINET fieldbus. In [LF07] just the PLC, switches and modules are considered for the simulation of a PROFINET network. TrueTIme [HCÅ03, CHO10] only uses the term network and nodes for a simulation - all other details are hidden in the simulation models and are not accessible for the user. In [MDFF06b], again just switches, IOs and PLCs are considered in the simulation.

This model also focuses only on a similar, smaller set of PROFINET specific elements. The first one is a *Switch*, which allows to structure the network into segments and collusion domains. Bridges are similar to switches with just a single connection other devices. They can be represented by specialized switches and are therefore neglected here. A repeater or adapter is used to extend an existing connection or to switch between different types of connections, like from copper to fiber. These elements are also neglected in this model, due to the fact, that they can be represented by special kinds of switches, too. Additionally, because these nodes increase the collusion domain of a network, they are often avoided in the design. Core elements remain the *PROFINETController* and *PROFINETDevice*s. PROFINET uses Buscoupler to connect modules (or slots) to the bus. In the approaches listed above, these couplers are often omitted or joined together with the modules. For the ease of use, their simplified models will be adopted here. Therefore, contain *PROFINETDevices* a set of modules represented by a given module size. *PROFINETDevices* function as multiplexer/demultiplexer for several IOs connected to the device. The *PROFINETController* needs to be modeled, but is usually also represented by, respectively is part of, the PLC. To model the specifics of a connection between controller, switches, and devices a *Connection* class is used. This class allows the developer to specify the length of the connection and the type (e.g. copper or fiber). Connections can be used between all elements that implement the *Connectable* interface.

Figure 5.17 shows this extension model which introduces the classes to model details of the PROFINET fieldbus. The *fieldbus* is extended by the *PROFINET* class which provides containment relations to various elements. To further specify correct fieldbus models, the two enumerations *Intervals* and *Modulsize* provide fixed values for setting up the cycle times for each *PROFINETDevice* and size of the used modules on each device.

Further details on how PROFINET works and the its devices can be found in Chapter 4. For the ease of use it it possible to combine several devices and their

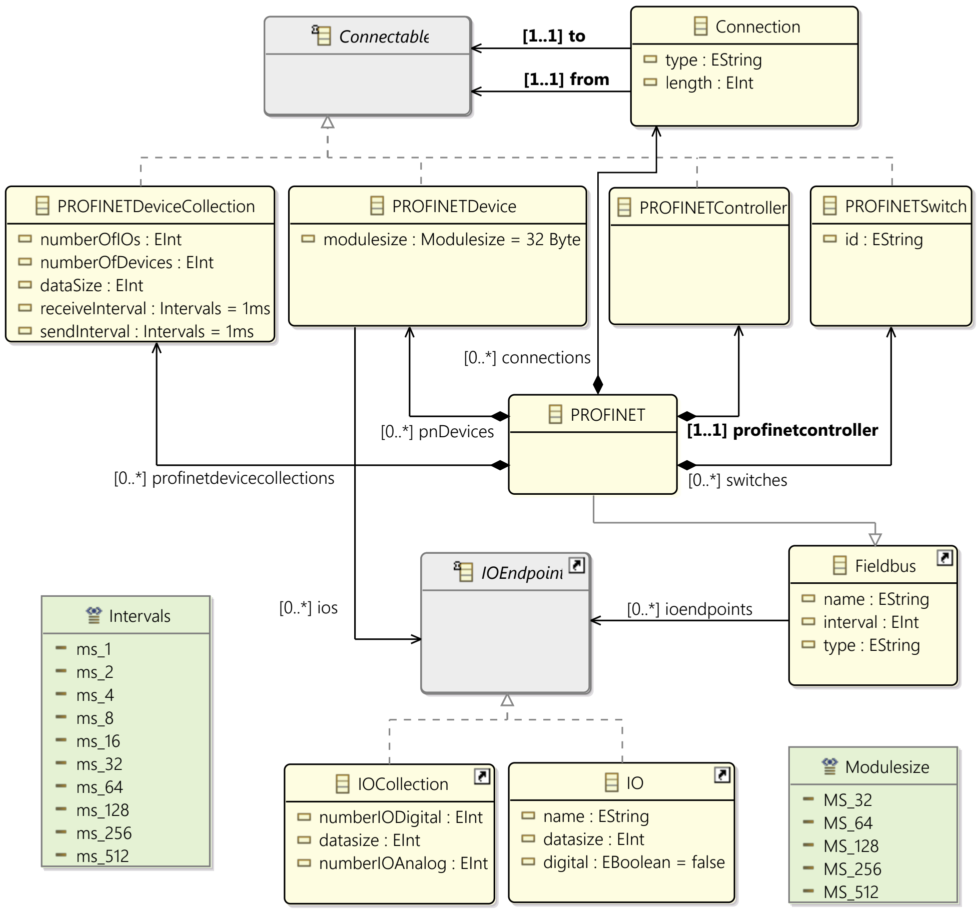


Figure 5.17: Classes overview for the PROFINET fieldbus

IOs in a single *PROFINETDeviceCollection*. The collection provides properties to specify the amount of devices and IOs, as well as the aggregated datasize and refresh intervals for all included devices. However, the provided model and its level of detail can further be detailed by additional components like gateways, repeater, firewalls, and so on.

INTERBUS TSM: The INTERBUS fieldbus [BM94] has been developed by Phoenix Contact and managed by the INTERBUS Club [Pro16b]. The fieldbus consists of several core elements. The Figure 5.18 shows an excerpt of the class diagram for the INTERBUS fieldbus. The *Interbus* class inherits its properties from the imported *Fieldbus* super type specified in the TIM. Each Interbus needs an *InterbusMaster* to control the fieldbus. This master is, similar to PROFINET, usually the PLC. Originating from the master is a remote bus that connects *BusTerminals* and *RemoteBusDevices*. *RemoteTerminals* are used to split a *local bus* or *loop* with *LocalBusDevices* from the remote bus. The *RemoteBusDevices* can also split a local bus but can additionally connect to IOs. Each *InterbusDevice* has product specific settings and properties. These properties include the maximum data size that can be transmitted via the device. To reflect this, a *deviceType* attribute has been added to model the product number or identification. The *Connection* class is used to connect the *InterbusMaster* and *InterbusDevices* with each other. To reflect the different bus types, the *connectionType* property can be used to detail the kind of connection that will be used.

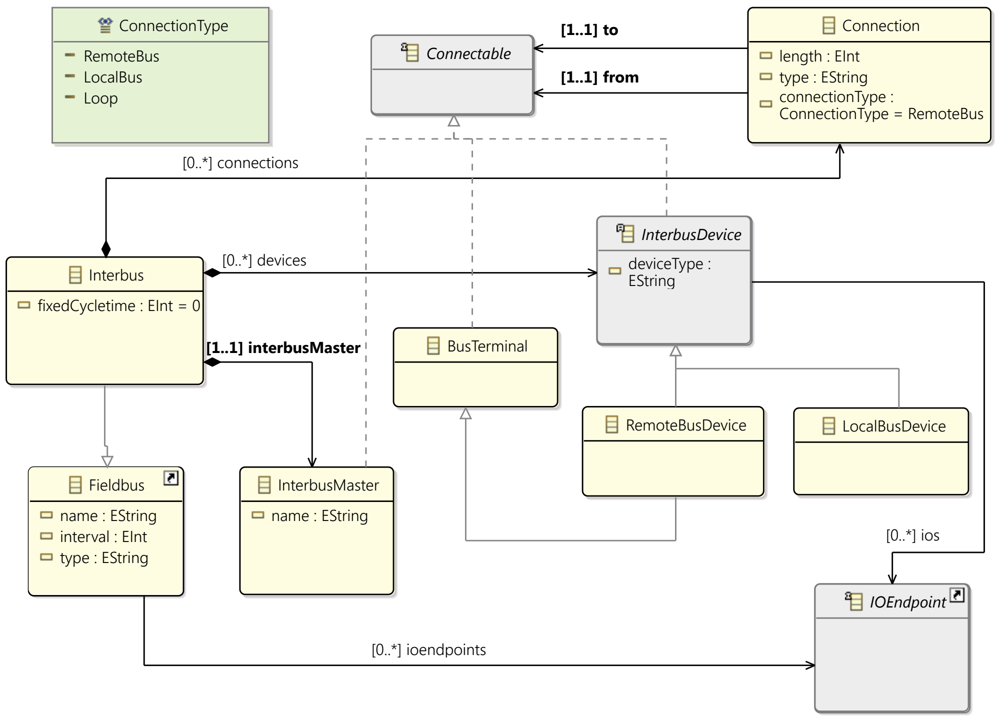


Figure 5.18: Classes overview for the INTERBUS fieldbus

As mentioned above, exist several alternatives to create a model for the Interbus fieldbus. The decision to use an enumeration to fix the type of connection will be much more easier to map onto existing CONSENS models than a dedicated connection or even a (remote/local/loop-) segment class. The goal of this thesis is to enable performance predictions in the early development stages. Therefore, such in-depth details about the topology and parameters of a fieldbus are not focused and should be handled by dedicated analysis tools in the domain specific development phases of the automation system. These tools might require additional information which could be added by a more complex TSM.

5.2 Development Process

In the first part of this chapter, the formal Automation Influence Model has been developed which captures the various influence factors. In this second part, the proposed process used model influence factors is developed. For this thesis, CONSENS has been selected as the Systems Engineering approach to model and analyze the system. Therefore, it is best to choose a CONSENS related development process instead of general automation specific or other Systems Engineering processes. The following section gives an overview over selected related work considering the development processes for automation systems. Afterward, the process to specify the influence factors is described in detail (Section 5.2.2).

5.2.1 Related Work

There are several approaches specifying processes for developing automation systems. Each has its own focus on, for example, requirements, hardware, or software. The following processes are listed here to give a short introduction to these processes before continuing to the proposed process.

- Vogel-Heuser et al.[VHSFL14, FEH$^+$12, VHBKF11] developed a SysML profile for modeling automation manufacturing software systems (MASP) called SysML-AT (SysML for Automation). They are focusing on four key aspects of a consistent MDE approach. The explicit modeling of functional and non-functional requirements, the modeling of automation hardware, the mapping of abstract machine functions of hardware, and an implementation that supports the generation of code.

  Figure 5.19 shows the process that does not require a strict sequential order. In the first step, the (non-)functional requirements (R1) are modeled. These describe the demand for production tasks, as well as real-time requirements for control loops. They are using the SysML concepts to model the requirements in a textual notation and link them to certain elements in the model. Using refinement relations, these requirements can be described in more detail later. In the second step, the software and machine functionality of the automation system (R2) is modeled. Automation functionality, like “angle measurement” describe the functional behavior of an automation software (AS) and use ports to define inputs and outputs. Based on the requirements (R1) and the results from R2, the automation hardware can be defined as *sensor*, *actuator* or *node*. The node is a resource which can later be used to deploy AS element. This is done in the fourth step, in which the hardware model and the software model are connected with each other. This deployment allows the desired generation of an initial software application and configuration which is IEC 61131-3 conform (R5).

  This approach extends the SysML language by automation specific elements to enable an automation engineer friendly development process. Discipline typical elements like sensors, actuators and nodes are added containing more specific properties which allow a generation of IEC code.

  The approach and development process, designed explicitly for automation systems, focuses on the initial creation and generation of the system. However, they do cover the analysis in early development phases.

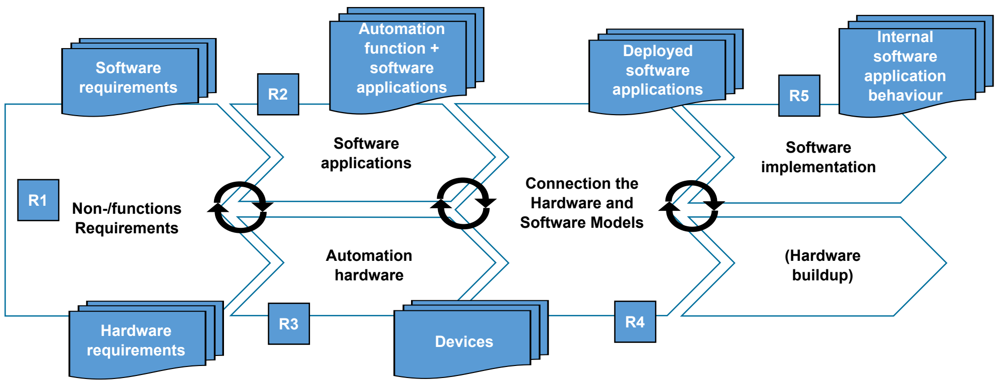

Software requirements
R2
Automation function + software applications
Deployed software applications
R5
Internal software application behaviour
Software applications
Software implementation
R1
Non-/functions Requirements
Connection the Hardware and Software Models
Automation hardware
(Hardware buildup)
Hardware requirements
R3
Devices
R4

Figure 5.19: System design phases (modified from [VHSFL14])

- Dubey presents an evaluation of software engineering methods in the context of automation application development (AAD)[Dub11]. She focuses on development processes based on common software engineering methods. None the less involves the development of AAD engineers from various disciplines. In her approach, she defines an automation application life cycle as shown in Figure 5.20. This process is based on their interaction with various practitioners and available literature.

  The primary phases are specified as follows. In the requirement and design phase, operational aspects of the AAD are collected. In this step, the gathering of requirements and the creation of the initial design of the system are combined into a single process design activity. She arguments that the line between gathering requirements and design the system is blurry and in some automation companies can not be strictly separated. This step fully covers the design of electrical plans, hardware construction, IO and task planning.

  The initial design of this first step is further refined in the following development step. The human machine interfaces (HMI) and control logic/control application are developed, based on the requirements and first sketches that contain for example lists of IOs and design diagrams.

  These two steps cover the development of the automation system. In step three are the necessary tests (module, integration, and factory acceptance test (FAT)) designed and carried out. In step four, the automation system is (disassembled), transported to the customer and assembled.

  The first two steps of this approach represent not an ideal development process, but the widely used sequential design of automation systems commonly referred as “throw-it-over-the-wall”. Each discipline is creating their development artifact in a specified order and deliver their artifacts to the next discipline, resp. department. For more information see 2.1.2 Development Process in [Rie14]. She presents a generally applicable development process of automation system applications, but without details that must be specified during certain development steps. Her process can

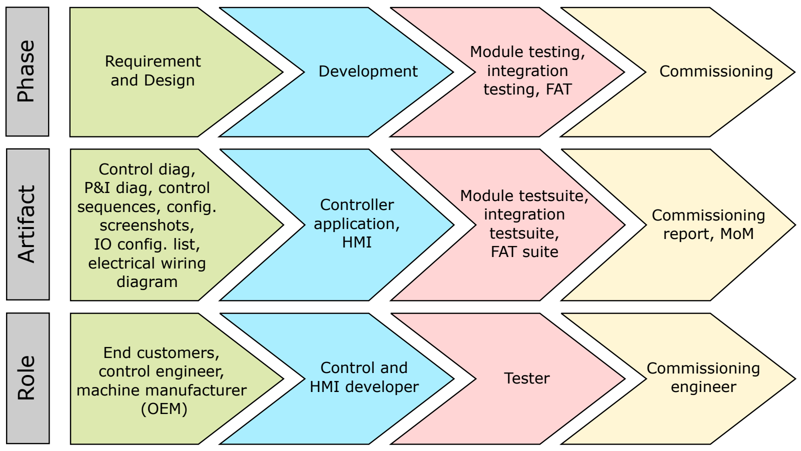

Phase
Requirement and Design
Development
Module testing, integration testing, FAT
Commissioning
Artifact
Control diag, P&I diag, control sequences, config. screenshots, IO config. list, electrical wiring diagram
Controller application, HMI
Module testsuite, integration testsuite, FAT suite
Commissioning report, MoM
Role
End customers, control engineer, machine manufacturer (OEM)
Control and HMI developer
Tester
Commissioning engineer

Figure 5.20: The Automation Application Development Lifecycle (modified from [Dub11])

be extended to cope the definition of influence factors and perform an early analysis of the system. However, due to the lack of a formal model and language, this would be an additional task for this thesis.

5.2.2 Proposed Development Process for Modeling Influence Factors

Since CONSENS has been selected as the Systems Engineering approach for this thesis, the most fitting development process for specifying the influence factors is specified in [HBM$^+$15, HBM$^+$16]. Their approach is based on [GRS14] and focuses requirements engineering and analysis in early development stages. They precisely specify the development steps, artifacts, and system engineering roles which match the desired approach to model the overall system structure. These system engineering roles are originally defined by Sheard [She96]. She presents twelve roles, each having distinct tasks during the development of a system engineering model. Out of these twelve roles, three roles fit the desired development process best and described in detail below.

The **requirements owner** (RO) translates the customer needs into specific requirements, understandable by engineers of the different disciplines involved in the development of the (automation) system. Artifacts produced by this role contain a functional architecture, capturing the need of the customer. The RO also assess the impact of requirement changes in the later phases of the development. The next role is the **system designer** (SD). An engineer taking this role is responsible for creating a high-level architecture of the system under development. The main focus is on the overall structure and the selection of components. *"Because of the complexity of projects employing system engineers, the emphasis tends to be on architecture, high-level design, integration, and verification, rather on low-level development"* [She96]. The inputs for this task are the previously defined functional architecture and specified requirements of the RO. The two roles (and steps) do overlap to a certain degree.

The last role is the **system analyst** (SA). The SA has to make sure that the system will meet the given requirements. With various analysis techniques – including performance prediction via simulation – this system engineering role makes sure that for example, the overall throughput is sufficient, outputs are generated in the desired quality, or the memory consumption is not exceeding the set limits. An important statement from Sheard is, that *"usually the more complex parts of the system need to be modeled in order to demonstrate that they will work properly ..."* [She96].

It is, therefore, necessary to model complex parts in more detail and anticipate/make assumptions about certain design specifics in the early development phases of the system design. For Sheard, the work of the system analyst role is either defined at specific points during or continuous throughout the development process/life cycle.

The development process refines the coarse Vee-Model which is based on [Ver04, HH08, GRS14] shown in Figure 5.21. The Vee-Model specifies three main development stages. The first phase, *system design*, is used to define the basic principles and concepts of the system under development. All disciplines that are involved in the development work together creating discipline-spanning Systems Engineering models.

This phase includes a wide range of steps like planning and clarifying the task, gathering and formalizing of requirements, creating the function hierarchy, developing/deriving the active structure and specifying the system behavior. The result of this phase is the so-called principle solution. This principle solution is used for the coarse structuring, agreement, and coordination of the involved disciplines and is, therefore, the input for the next development phase, the *discipline specific development*.

In this phase, the involved disciplines create more detailed models and specifications of the system under development. Each discipline uses also a wide range of specialized tools and languages for this task. Examples are UML [RJB04] for the software engineering discipline, Matlab/Simulink [Mat16] for the control engineers, EPLAN [EPL17] for the electrical engineers and SOLIDWORKS [Das17] for the mechanical construction.

The results of this separated development are combined in the *system integration* phase to create the system. Depending on the thoroughness and quality of the requirements and systems engineering models from the system design phase, this integration can either be smooth or result in costly changes. To check whether the requirements and designs are correctly realized a continuous testing and verification of the given requirements is necessary.

The Figure 5.22 shows the development process further detailing the system design phase. It is based on the process developed by Holtmann [HBM^{+}15] et al. and extends it by a specific step for specifying the influence factors or other

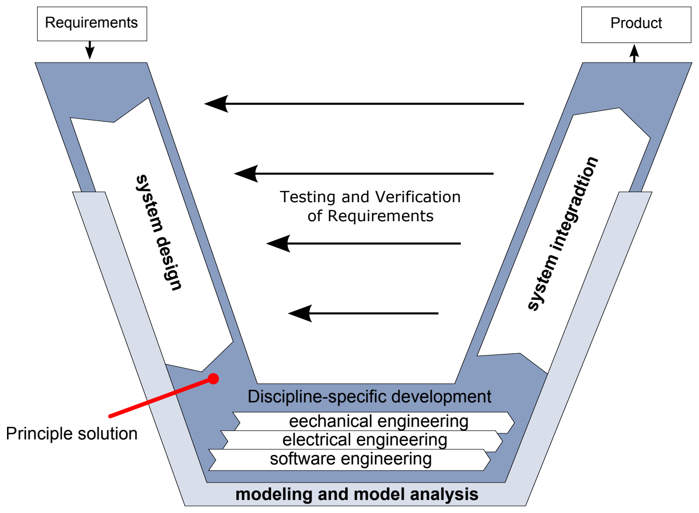


Figure 5.21: Vee-Model process for system development (modified from [Ver04, HH08])

performance related information in the CONSENS models. According to the BPMN [Obj11a], manual steps are indicated by a hand in the upper left corner of a step and (semi) automatic steps by a gear symbol. The first step is to

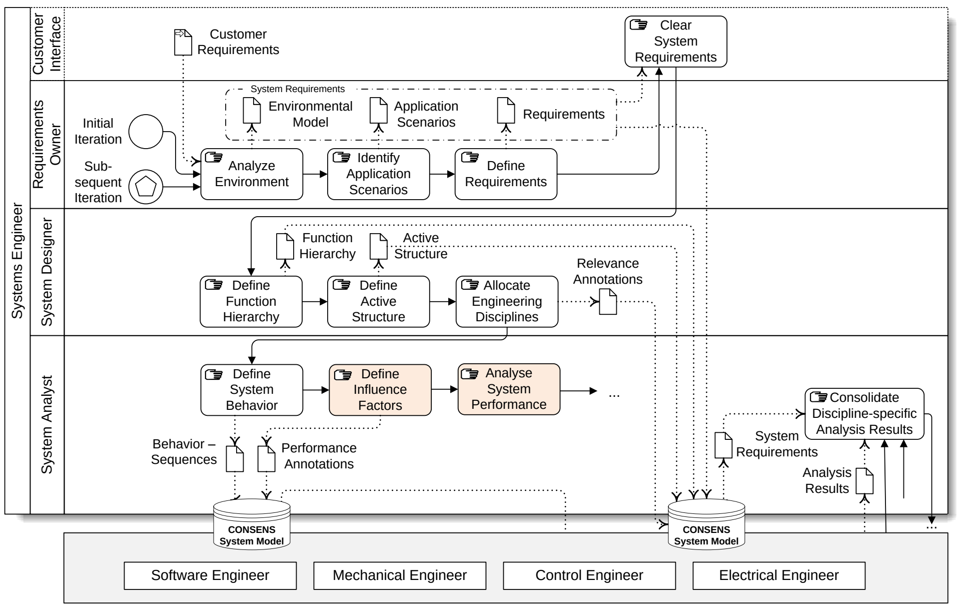


Figure 5.22: Proposed development process based on [HBM+15]

capture the customer requirements. For this task the system engineering role named *customer interface* (see [She96]) is responsible. The requirements are the input for the subsequent step, in which the requirements owner analyzes the environment of the system and captures it with the CONSENS environment model. Afterward, the output of **Identify Application Scenarios** is a list of application scenarios, each roughly describing common operation modes and behavior of the system. Each scenario focuses on a specific (technical) situation.

Another task for the requirements owner is the definition of requirements (see step **Define Requirements**). The partial model *Requirements* contains a collection of requirements, each uniquely identifiable and referable throughout the partial models. These requirements specify the expected/desired behavior of the system under development and therefore are an essential basis for the validation and verification in the following development phases. Afterward, the captured requirements must be approved by the customer, respectively the customer interface role. This is done in the **Clear System Requirements** step.

In case all requirements are correctly specified, the work of the requirements owner ends until the next iteration of the development process and the system designer starts by defining or refining the function hierarchy (**Define Function Hierarchy**). Functions of the system fulfill the defined requirements and can be broken down into more detailed subfunctions. The hierarchy starts with a single function specifying the overall task of the system.

The following development step *Define Active Structure* is used to create or update the CONSENS specific model active structure. The active structure defines the internal composition and relationships between elements. More details on the CONSENS active structure and its various model elements can be found on the foundation chapter 2.2.1.

Up to this step, the active structure and its elements are used by all involved engineering disciplines to identify relations or interfaces to the various parts of the system. But in the discipline specific development phase of the Vee-Model, not all elements are considered by each discipline. Therefore the engineering disciplines are allocated to each element with relevance annotations [HSST13, Rie14]. An example of such a relevance annotation is the energy supply unit. This unit provides electrical power to most parts of the system and is important for the electrical and mechanical engineering discipline. The software and control engineer are not influenced by changes or modification and are therefore not allocated to this active structure element.

After the engineering disciplines have been allocated, the system analyst can define the behavior of the system (see step **Define System Behavior**) by using one or more behavior models provided by CONSENS. They include *State Machines*, *Activity Diagrams* and behavior sequences model with *Sequence Diagrams*. In [HBM+15], the authors use a formal extension of sequence diagrams called modal sequence diagrams, to model specific requirements. Given the detailed behavior of the system, its structure, and environment, they are able to use a simulation to assure that the system meets the given requirements for a certain situation. Up to this point, the proposed development process is similar to the one presented by Holtmann. However, to check whether the system does not only behave like required, but also provide the performance necessary to fulfill the requirements, further analysis must be carried out.

This thesis specifies two additional steps to extend the existing development process. In the step **Define Influence Factors**, the active structure is annotated by automation specific influence factors. Each factor is providing additional information based on the influence factors identified in Chapter 4.

Figure 5.23 is giving an overview of these steps which will be detailed in the following. The first three steps can be executed in any order due to the fact, that definitions or changes to existing influence factors do not affect each other. The following steps are depending on the data of one or more previous steps.

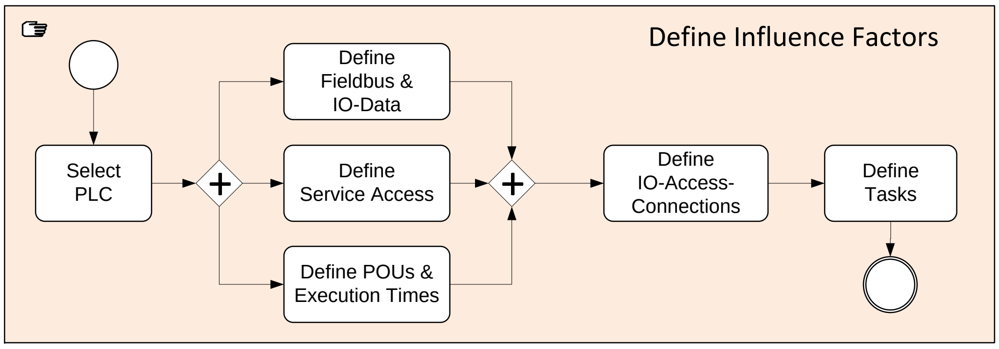

Define Influence Factors
Select PLC
Define Fieldbus & IO-Data
Define Service Access
Define POUs & Execution Times
Define IO-Access-Connections
Define Tasks

Figure 5.23: Detailed sub steps of 'Define Influence Factors'

Select PLC The first step is to specify the PLC that will be used for an analysis of the overall system. This PLC must support the selected fieldbusses, services, and task types. Usually, the active structure already includes a system element to represent the PLC. This element needs to be annotated and the corresponding properties, like name and type, specified accordingly. Depending on the underlying analysis approach, the engineer selects an available PLC from a given set that is covered by the analysis implementation. For this thesis, the available PLCs are provided by load profiles modeled with Palladio (see Chapter 6). If previous analysis iterations have already been made, a more fitting PLC might be exchanged with the existing one (e.g. the overall utilization was too high). Other reasons for switching a PLC could also be a different fieldbus that must be supported or a new layout of the automation system.

Define fieldbus & IO-Data In this step, all information related to the fieldbus and its specific settings is specified. This includes the fieldbus, sensors, actuators, devices and further settings. Usually one or more fieldbusses are used in an automation system. The different kinds of bus systems need to be captured and represented in the active structure. Second, the available sensors and actuators must be identified. These elements are either already modeled in the active structure or need to be approximated by marking elements as an IO collection. Also, each fieldbus has specific settings that need to be set up. These could be for example the cycle time in which data is exchanged between the PLC and the IOs. Depending the on the selected fieldbus and the current modeling depth, it might be necessary to specify the distances (length) between the PLC and different IO devices connected via the bus.

Define POUs & Execution Times An important step is to specify which Programs, Functions, and Function Blocks will be used and what their timing or execution properties are. Often, automation systems are not developed from scratch but are reused from similar, existing projects. This allows the developers to make assumptions on how the POU will behave and what execution times can be expected. In case a completely new POU must be developed, the properties must be roughly (over)estimated. Estimated execution times of a Program or Function can also be used as a requirement to adhere. Software developers are forced to check whether their POUs do not take more time to execute than specified. If the requirements can not be met, the different disciplines have to be informed and new solutions or requirements agreed on.

During the discipline-specific development phases, a software engineer can refine or detail the actual execution times on a POU. This allows him to correct the estimation on a previously defined POU or to add additional POUs that are necessary for the functionality of the automation system. Again, changes on any influence factor must be propagated into the involved disciplines.

Define Service Access The Environment model and the Active Structure already include elements that access data of or interact with services. To model such an access, the accessing element will be extended by additional information on how often a service access occurs. It must also be specified, which service is accessed and what kind of access is performed (VariableAccess, FileAccess, or OperationAccess). This applies to all elements that access any service, including System Elements and Environment Elements.

Define IO-Access-Connections After the fieldbus is set up and the different Programs and Function Blocks are known, the access of IOs from the PLC, respectively the different POUs, can be specified. The POUs read data provided by sensors and send commands to the actuators over a fieldbus. Therefore, it is necessary to know which programs accesses which IOs. After this information is added to the model, payload sizes and – depending on the used fieldbus – cycle times can be computed.

Define Task One of the last steps is to set up the different types of tasks on the PLC. Each task can execute one or multiple Programs that have been specified in the previous steps. The selection of a task type and the configuration of cycle times can be further restricted by automation system (e.g. production throughput) or Program or Function Block specific requirements (e.g. minimum/maximum execution time for an algorithm).

Depending on the refinement iteration of the system model, some of these steps could also be skipped and just parts of the model can be updated. After specifying the different influence factors and their parameters, the analysis and interpretation of the results must be performed.

This is done in the subsequent process step **Analyze System Performance**, which provides the system analyst with information about the possible utilization and performance of the automation system. Various tools and techniques can be used to investigate different aspects of the system. A selection of them are introduced in Chapter 6 along with an in-depth example based on the Palladio Component Framework.

5.3 Summary

In this chapter, several aspects are discussed and developed. It introduced the Automation Influence Model to formally capture the influence factors identified in Chapter 4. This formal model provides well-formed syntax and semantics, making it (automatically) process- and analyzable. This is a necessary prerequisite to run performance simulations based on the influence factors the automation developers provide. To develop this formal model, several questions had to be answered first. The required modeling depth to capture the influence factors (Section 5.1.2), a follow up discussion about topology dependent and independent modeling (Section 5.1.5) and how the utilization of a PLC can be described hardware independently (Section 5.1.4). Afterward, the parts of the formal Automation Influence Model are introduced (Section 5.1). This model captures all influence factors, their properties and relations between them and other Systems Engineering elements. In the second part of this chapter, a development process and its steps is presented. It is based on an existing development process [HBM$^+$15] and extends it by two steps. The development process is used by developers in the system analyst role, to annotate existing Systems Engineering models with the identified influence factors. The resulting proposed process is explained in section 5.2.2.

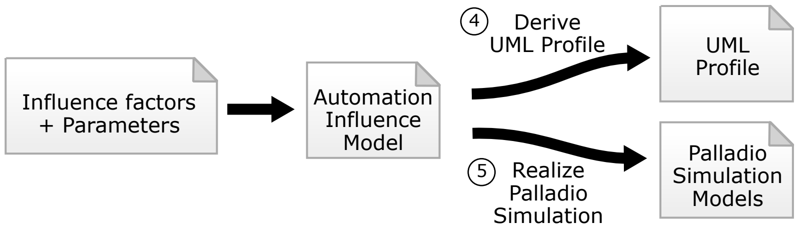
It is now possible to use a generally applicable, CONSENS-based development process to build up a formal model that can be used for further performance analysis.

In the following Chapter 6, the developed method is realized/implemented by creating a UML profile that allows the annotation of existing Systems Engineering models. Additionally, a fitting analysis approach is selected and the structure, generation, and use of input models for this analysis is described.

CHAPTER 6

Realization of Modeling and Analysis

In Chapter 4, the different influence factors that impact the performance of an automation system are identified. In Chapter 5, these factors are captured in a formal model called Automation Influence Model. To be able to evaluate the contribution of this thesis, a tool for annotating existing Systems Engineering models with the influence factors and an subsequent analysis to conduct a performance prediction must be developed.



Influence factors + Parameters
Automation Influence Model
4 Derive UML Profile
UML Profile
5 Realize Palladio Simulation
Palladio Simulation Models

Figure 6.1: Creating the UML profile and Simulation models

In the first part of this Chapter, the formal model, respectively the influence factors, are mapped onto an UML profile (Step 4 in Figure 6.1). This profile is used to annotate existing CONSENS Systems Engineering models and extends the SysML4CONSENS profile [KDHM13, IKDN13]. All profiles and models are created for and with the Papyrus Modeling environment [Ecl17c] as part of the Eclipse IDE [Ecl17a].

In the second part of this chapter, approaches and tools for the performance prediction of an automation system are evaluated (Step 5 in Figure 6.1). As a result of this evaluation, the Palladio Modeling Framework has been selected for this thesis. It provides means to model, simulate, and analyze an automation system. The models necessary for the simulation of the different influence factors and their parameters are introduced in the following sections. To improve the possibly error prone and time-consuming process of creating the simulation models manually, different model-to-model transformations are introduced which carry out this task automatically.

This chapter is structured as follows. First, an overview of the different models and transformation steps is given in Section 6.3. Afterward, the details of the UML profile that extends SysML4CONSENS are introduced. Section 6.2 gives a short introduction to the Palladio models and highlights selected details. Finally, in Section 6.3.2, the different transformations are specified that are used to transform the CONSENS Systems Engineering models into Palladio simulation models.

6.1 UML Profile for Modeling Influence Factors

In Chapter 5 the formal basis to model the influence factors and their parameters has been set up with the Automation Influence Model (AIM). This model is now mapped to a UML profile to be able to extend existing CONSENS models. Most of the elements and properties of this profile are 1:1 representations of the AIM elements. However, some changes are made to better parse and transform the annotated model into Palladio simulation models. Additionally, these changes improve the overall modeling work flow for the system analyst and the integration into the Papyrus tool.

To create CONSENS models currently two free of charge approaches can be used. The first is a set of Microsoft Visio shapes. However, Viso based models are difficult to access via external tools, making them a poor choice as input models for further tools in a toolchain. The second option is to use the SysML4CONSENS UML profile [KDHM13, IKDN13]. This profile extends SysML and adds CONSENS specific stereotypes. It enables developers to create CONSENS models that can be easily parsed, analyzed, and with various techniques (e.g. QVTO [Obj11b]) transformed into other models.

There are already a wide range of UML profiles available that provide automation specific or performance related stereotypes. These profiles are listed in the related work sections of Chapter 4 and 5. MARTE [Obj06] is the defacto standard for annotating performance and timing information for embedded realtime software. However, the profile is quite complex, contains a plethora of stereotypes, and is designed specifically for the in-depth development of software. For a high-level annotation of Systems Engineering models MARTE is not a practical choice. It is possible to reuse elements from MARTE like the ArrivialPatterns, base types, units, or distribution functions. But this would add additional dependencies to the profile and increase the overall modeling complexity. The SPTP profile [Obj05] is not considered here, since MARTE is an inofficial successor of this profile and already covers/includes most modeling features. Vogel-Heuser et al. also provide a UML profile based on SysML [VHBKF11]. This profile is used to annotate automation system specific elements like sensors and actuators. However, the profile does not provide sufficient information for a performance prediction. Extending this profile would reduce the overall modeling effort just marginally but induce new dependencies to other profiles. The UML-RT profile [Sel98, KHCD17] does not provide sufficient stereotypes to model different arrival times, randomness or the possibility to model hardware dependent or independent resource usages. It can be used to specify WCET and time constraints, but adding the missing model elements and mapping these to the UML-RT concepts would be too much overhead and add additional dependencies to the AIM profile, too. For these reasons, a new profile has been developed which is detailed in the following subsections.

When to use UML profiles instead of meta models is a question depending on the purpose and future use of the created models [Des00, SW06, SVC06]. For this thesis, the profile mechanism from the UML specification has several advantages. First, is CONSENS already available as a profile. Extending this profile with additional stereotypes will reduce the overall modeling effort. Most of the necessary elements and their relations are already present in the UML, SysML, or SysML4CONSENS classes and stereotypes. Figure 6.2 sketches the relations between the different models. The UML elements provide the basis for the SysML profile. This profile adds new elements and further constrains the use of UML classes and diagrams (see Chapter 2). SysML4CONSENS (just CONSENS in the figure) does the same with the SysML profile. The AIM profile introduces new stereotypes that extend or reference elements in the SysML4CONSENS, SysML, and UML. This approach wont induce further dependencies, because CONSENS4SysML cant be used without the SysML profile and all profiles make use of the basic UML classes.

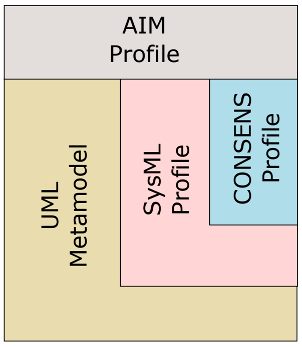


Figure 6.2: Relations between UML, SysML, CONSENS, and AIM

Second, there already exists a broad range of UML compliant tools, that are capable to view and edit models like Papyrus [Ecl17c] or Enterprise Architect [Spa17]. A third advantage is the ability to create additional transformations much easier using standard frameworks like QVTO. For example can Papyrus models with applied stereotypes be transformed into MATLAB/Simulink models. Or the annotated information from the CONSENS models can be used to create initial UML models for the software engineer. And last, if the performance specific information is no longer needed, the profile can be easily and side effect free removed. Stereotypes only add information or constrain the original model. When removing a profile, the original model will still be valid.

The AIM profile does not introduce new kinds of diagrams. It uses the already existing diagrams to model structural or behavioral information. The SysML4CONSENS profile uses SysML Block Definition Diagram (BDD) and Internal Block Diagram (IBD) to specify the active structure. To model behavior, the Activity and State Diagram from the UML specification are used.

Diagrams to model or allocate software onto (hardware) resources are not used. An introduction to the SysML4CONSENS profile can be found in Chapter 2.

The majority of classes that are extended by the AIM stereotypes are specified in the UML meta model. This is a design decision to make the profile as generally applicable as possible. With only a few changes, the profile can be set to extend only SysML or even just UML. However, the focus of this thesis is the creation of a profile that can be used to annotate CONSENS models specified via the SysML4CONSENS profile. To do this, the AIM profile extends several UML classes like Class, Property, Activity, Operation, CallOperationAction, OpaqueOperationAction, or Parameter. Constraints to certain CONSENS specific elements are made with OMG's Object Contraint Language [Obj17] or by setting explicit references to the SysML4CONSENS stereotypes. These are usually *SystemElement*, *EnvironmentElement*, and *InformationFlow*. Another advantage of using SysML base classes and diagrams for the AIM profile is the native support of the Papyrus editor. It allows a simpler and more intuitive handling of SysML Blocks than UML classes. However, this has no impact on the semantics of the profile and solely improves the modeling work flow for the developer creating annotated CONSENS models.

Figure 6.3 sketches the overall structure of the AIM profile. Common elements to specify, for example, resource usage, time units, or file sizes are contained directly in the AIM package. The UML profile is structured after the Automation Influence Model. All influence factors are based on the elements of the formal model. The model, as well as the UML profile, are structured after the different categories of influence factors. The four sub-packages are created to group elements by their primary use. Function Blocks and Programs are specified in the pou-package, fieldbusses and their elements in the io-package, the different access kinds for services in the service-package, and task related elements in the the task-package. The following subsections detail the profile and give selected examples of their usage.

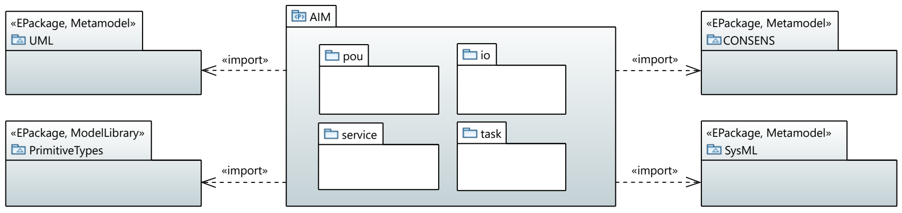


Figure 6.3: Overview of the AIM Profile and packages

6.1.1 AIM Profile Root Package

The root package contains common elements that are used throughout the profile. Figure 6.4 shows an overview of these elements. The stereotype *PLC* is used to specify the PLC and to provide all necessary properties for the analysis phase. For this, the name of the PLC and its vendor is sufficient. An analysis tool needs to lookup the corresponding simulation models for this PLC. To model the IPTraffic no explicit stereotype has been created, but all properties moved to the PLC itself. This should simplify the modeling process. These properties are used to specify the amount and size of incoming IP packages. For each parameter, the value kind must be set up. The *ValueKind* is an enumeration with the literals fixed, bounded, randomset, or random interval. This is used to parse the corresponding literals (*numberOfPackagesLiteral*, *sizeOfPacketsLiteral*) correctly. It would also be possible to omit the valueKind and identify the literal just by parsing it. The stereotype PLC extends the class, allowing its use in an UML, SysML and CONSENS model.

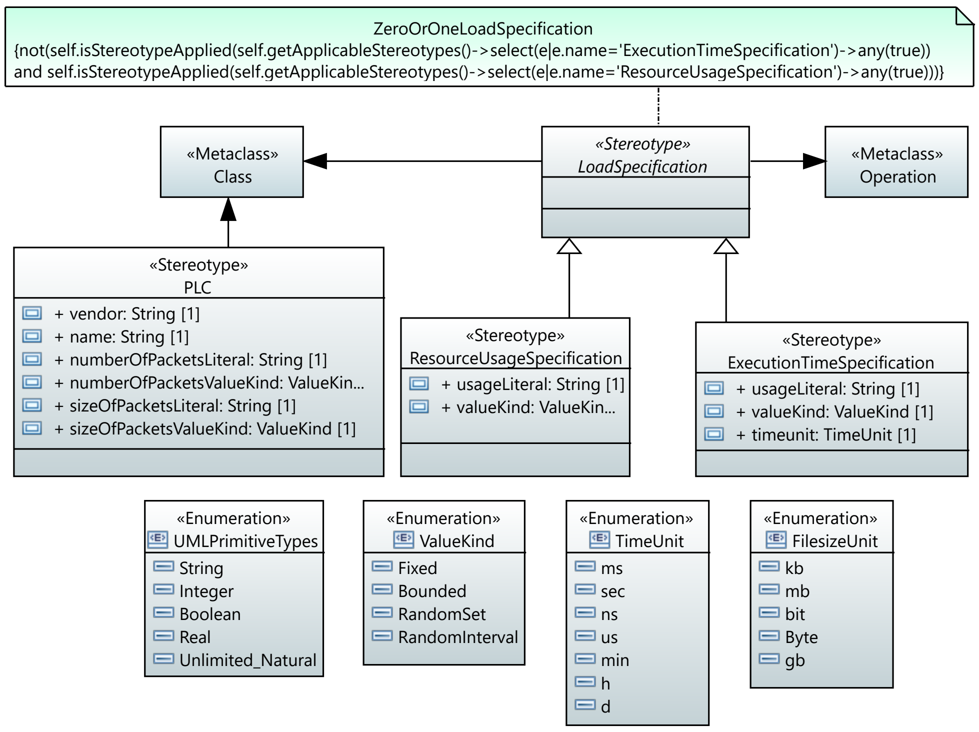

ZeroOrOneLoadSpecification
{not(self.isStereotypeApplied(self.getApplicableStereotypes()->select(e|e.name='ExecutionTimeSpecification')->any(true))
and self.isStereotypeApplied(self.getApplicableStereotypes()->select(e|e.name='ResourceUsageSpecification')->any(true)))}
«Metaclass» Class
«Stereotype» LoadSpecification
«Metaclass» Operation
«Stereotype» PLC
+ vendor: String [1]
+ name: String [1]
+ numberOfPacketsLiteral: String [1]
+ numberOfPacketsValueKind: ValueKin...
+ sizeOfPacketsLiteral: String [1]
+ sizeOfPacketsValueKind: ValueKind [1]
«Stereotype» ResourceUsageSpecification
+ usageLiteral: String [1]
+ valueKind: ValueKin...
«Stereotype» ExecutionTimeSpecification
+ usageLiteral: String [1]
+ valueKind: ValueKind [1]
+ timeunit: TimeUnit [1]
«Enumeration» UMLPrimitiveTypes
String
Integer
Boolean
Real
Unlimited_Natural
«Enumeration» ValueKind
Fixed
Bounded
RandomSet
RandomInterval
«Enumeration» TimeUnit
ms
sec
ns
us
min
h
d
«Enumeration» FilesizeUnit
kb
mb
bit
Byte
gb

Figure 6.4: AIM profile root package

Another important element is the abstract *LoadSpecification* which is specialized by the *ResourceUsageSpecification* and the *ExecutionTimeSpecification*. They extend the UML *Class* as well as the UML *Operation* to allow a single specification of the stereotype and its use for multiple influences like a Program, Function, or ServiceAccess. The *ResourceUsageSpecification* has two properties to set up the amount of CPU resource that will be consumed during the call, execution, or access. Like the property pairs on the PLC is *valueKind* used to specify the kind of access, allowing to easily parse the second *usageLiteral* property for the values.

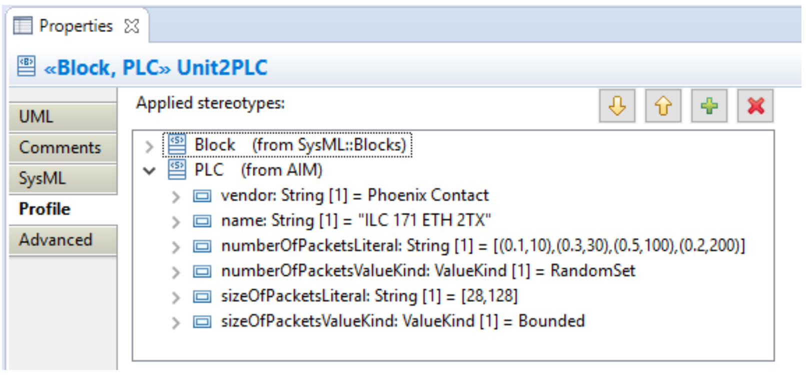
The *ExecutionTimeSpecification* is used to define a time it takes to execute for example a Program, Function Block, or ServiceAccess. Again, the *value-*

Kind restricts a String that can be specified via the property *usageLiteral*. An additional property *timeunit* is used to set the time scale.

ResourceUsageSpecification and *ExecutionTimeSpecification* are two distinctive elements that can be applied to the same element in a model. This must be prohibited by an OCL constraint, allowing either one kind of *LoadSpecification* or none.

The remaining stereotypes are enumerations that will be used in more than one sub package and are therefore defined directly in the root model. These are the *FilesizeUnit* to easily select a fitting file size and the UMLPrimitiveTypes. They are used to select a primitive type more easily than browsing through the model and picking the UML datatypes.

Figure 6.5 shows a screenshot taken from the Papyrus editor. It shows the applied stereotypes Block from SysML and PLC from the AIM profile on the base UML Class. By applying the stereotype *PLC* to the Class/Block, the developer has now formally set up one PLC for the automation system. It is now possible to further specify properties like the vendor and type of the PLC. These two properties are mandatory to later create the simulation models for the performance prediction. The other properties can, but need not be specified and are used to model additional load that is induced by background IP traffic.



Block (from SysML::Blocks)
PLC (from AIM)
vendor: String [1] = Phoenix Contact
name: String [1] = "ILC 171 ETH 2TX"
numberOfPacketsLiteral: String [1] = [(0.1,10),(0.3,30),(0.5,100),(0.2,200)]
numberOfPacketsValueKind: ValueKind [1] = RandomSet
sizeOfPacketsLiteral: String [1] = [28,128]
sizeOfPacketsValueKind: ValueKind [1] = Bounded

Figure 6.5: Additional properties provided by the PLC Stereotype

6.1.2 POU Package

This package contains all stereotypes used to model automation related software. This covers Programs, Function Blocks, and Functions. *Programs* and *Function Blocks* are derived from the abstract POU stereotype. They do not contain further attributes due to the fact, that they are only used to further annotate existing *Classes*, respectively SysML Blocks or CONSENS SystemElements. The *Function* stereotype extends the UML *Operation* to annotate an automation system software function. Via the UML *Operation* none or multiple Parameter can be modeled. Therefore, the *FunctionParamter* stereotype extends the UML *Parameter*. It is possible to add further properties to the Program, Function Block, and Function to reference contained, AIM specific, model elements. This could simplify the access to elements (e.g. for OCL or QVTO queries) but would also add redundant information which is already been specified by the UML properties. The *LoadSpecification* in the diagram has been imported, to visualize that *ResourceUsage* and *ExecutionTimes* can be applied to Programs, Function Blocks and Functions.

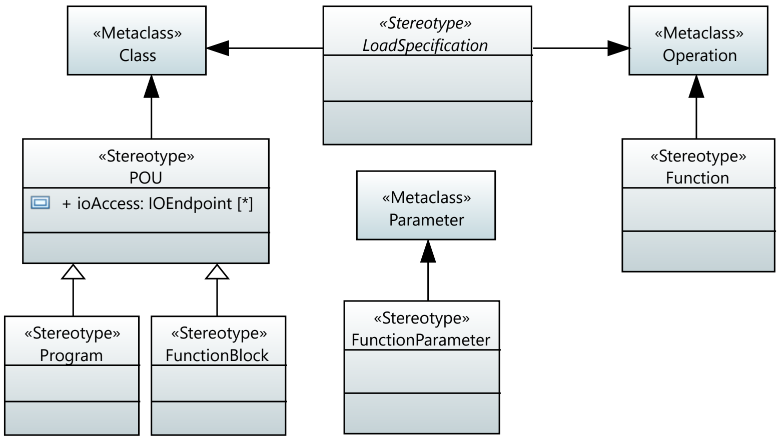
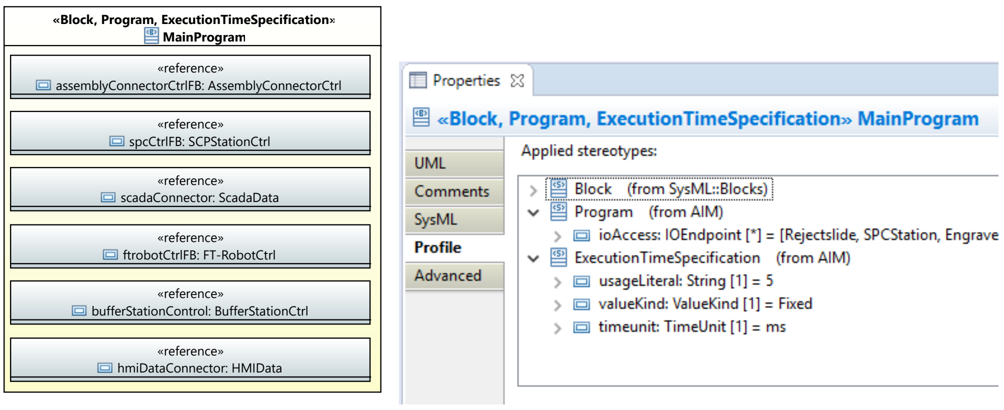


Figure 6.6: Definition of Stereotypes to model the automation software

Figure 6.7 depicts on the left side (6.7a) the different parts of the *MainProgram*. These parts – not shown in this diagram – are typed by SysML Blocks (or UML Classes) and extended by the Function Block stereotype. On the right side (6.7b) is a screenshot that shows the properties of the MainProgram. The Program has a fixed execution time of 5ms which has been specified via the ExecutionTimeSpecification stereotype. Additional information is provided by listing the different IOs or IOCollections, which the MainProgram will access, respectively, which IO information will be used by the Program.

6.1.3 Task Package

This small package contains the definition of three task types as depicted in Figure 6.8. All of these tasks are derived from the abstract stereotype *Task*, which defines a basic *priority* and *coreAffinity*. It extends the UML *Class*, which allows the use in SysML and CONSENS models. The hierarchical structure of Tasks containing programs and Function Blocks can be modeled best with the established UML Class and property (respectively part) mechanisms. The *CyclicTask* further provides properties to set the *cycleTime* and its time scale via the *timeunit*. The *IdleTask* has no additional information to specify, due to its highly PLC and vendor specific implementation. The last stereotype is the *EventTask* that will be triggered by internal or external events specified in Programs or the PLC settings. The *triggerKind* uses the introduced *ValueKind* enumeration to define how the *triggerLiteral* String will be parsed.


«Block, Program, ExecutionTimeSpecification»
MainProgram
«reference»
assemblyConnectorCtrlFB: AssemblyConnectorCtrl
«reference»
spcCtrlFB: SCPStationCtrl
«reference»
scadaConnector: ScadaData
«reference»
ftrobotCtrlFB: FT-RobotCtrl
«reference»
bufferStationControl: BufferStationCtrl
«reference»
hmiDataConnector: HMIData
Properties
UML
Comments
SysML
Profile
Advanced

«Block, Program, ExecutionTimeSpecification» MainProgram

Applied stereotypes:

- Block (from SysML::Blocks)
- Program (from AIM)
  - ioAccess: IOEndpoint [*] = [Rejectslide, SPCStation, Engrave
- ExecutionTimeSpecification (from AIM)
  - usageLiteral: String [1] = 5
  - valueKind: ValueKind [1] = Fixed
  - timeunit: TimeUnit [1] = ms

(a) MainProgram parts (b) MainProgram properties

Figure 6.7: Specification of the Program *MainProgram*

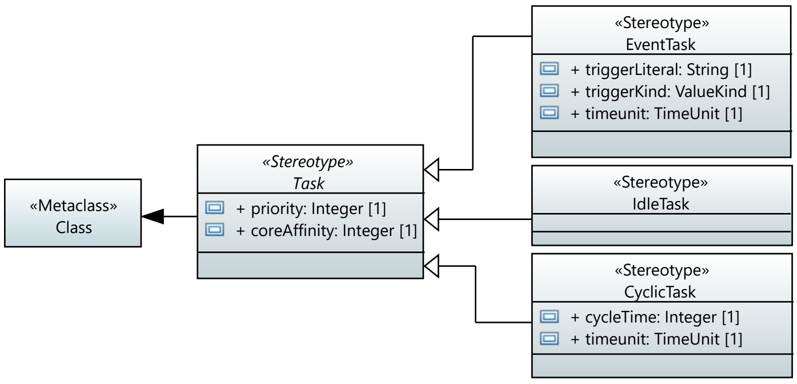
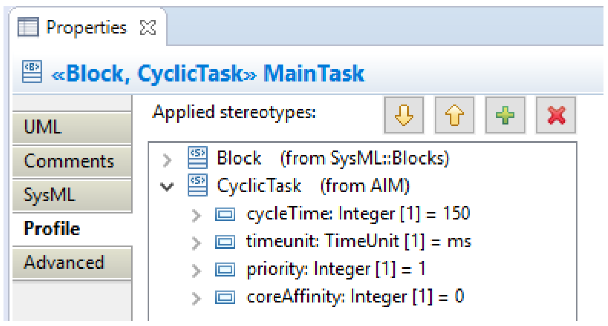


Figure 6.8: Stereotypes to define different kinds of tasks

An application of task stereotypes is shown in Figure 6.9 . Like the Programs or Function Blocks is it necessary to first create Classes and extend them with a Task stereotype. Afterward, a part of this type needs to be added to a PLC to finish the specification of tasks. The figure shows the properties tab of the Papyrus editor and the available parameters that can be specified. In this case, a cyclic task with a cycle time of 150 ms and the highest available priority is modeled.

6.1.4 Service Package

Figure 6.10 shows the content of the service package. The core element of this package is the *Service* stereotype, which is used to annotate an UML Metaclass to indicate that the extended element is a service provider. The service provider can be accessed by other model elements by using the *ServiceAccess*, which references the *Service*. The three different types of ServiceAccess are *VariableAccess*, *FileAccess*, and *OperationAccess*. The first two extend the UML *OpaqueAction* Metaclass, which allows the specification of these access types



«Block, CyclicTask» MainTask

Applied stereotypes:

```
> Block    (from SysML::Blocks)
v CyclicTask    (from AIM)
    > cycleTime: Integer [1] = 150
    > timeunit: TimeUnit [1] = ms
    > priority: Integer [1] = 1
    > coreAffinity: Integer [1] = 0
```

Figure 6.9: Properties of the *MainTask*

in behavioral diagrams using actions. The last one extends the *CallOperationAction* to reuse the standard UML semantics for executing operations. For the same reason do *SerivceOperation* and *ServiceOperationParameter* just provide stereotypes for *Operation*, respectively *Parameter*. It is a more intuitive and effective approach to reuse the basics of the UML standard for specifying operations, their parameters, and operation executions than to provide new, similar concepts. Additionally, do most of the UML complaint modeling tools support this kind of use and represent them accordingly.

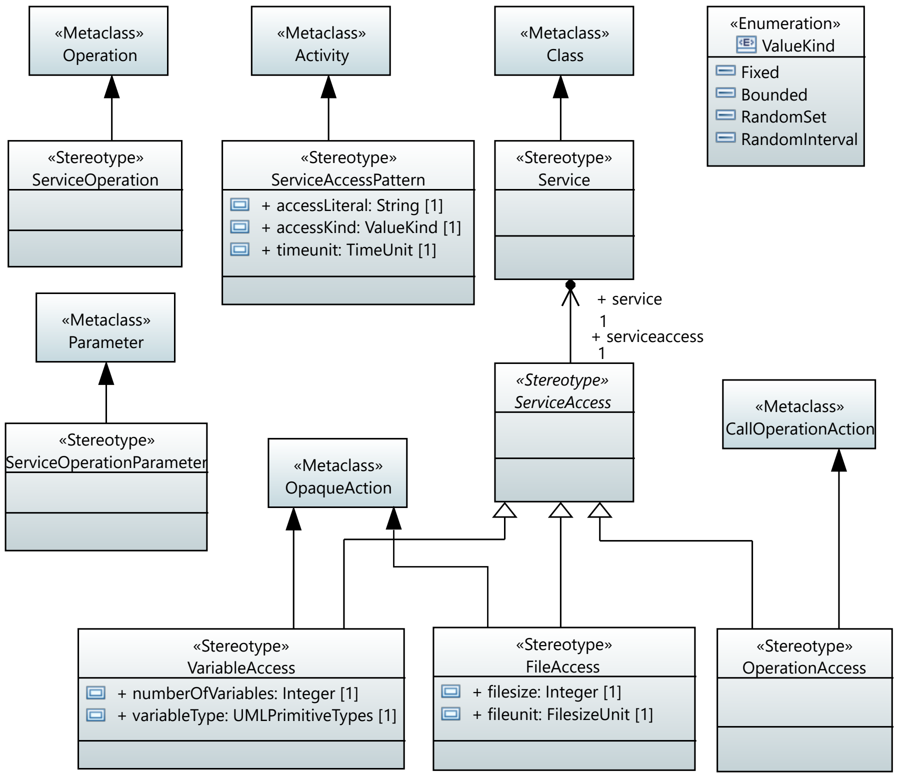
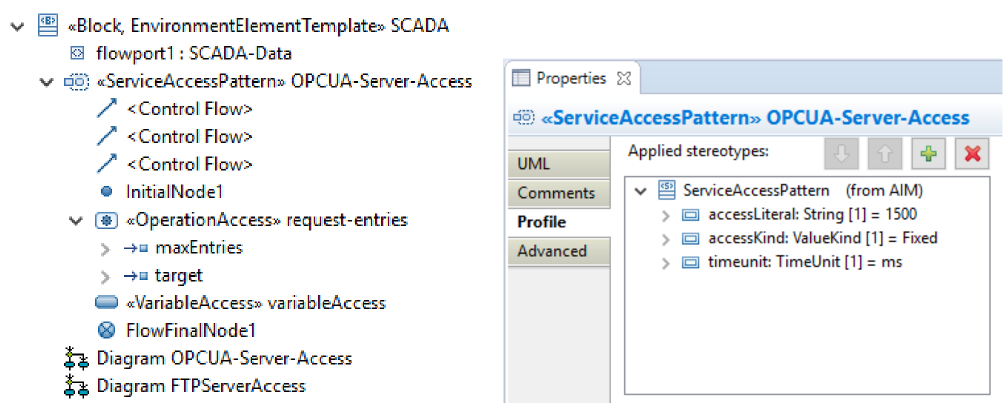

«Metaclass» Operation
«Metaclass» Activity
«Metaclass» Class
«Enumeration» ValueKind
Fixed
Bounded
RandomSet
RandomInterval
«Stereotype» ServiceOperation
«Stereotype» ServiceAccessPattern
+ accessLiteral: String [1]
+ accessKind: ValueKind [1]
+ timeunit: TimeUnit [1]
«Stereotype» Service
+ service
1
+ serviceaccess
1
«Metaclass» Parameter
«Stereotype» ServiceAccess
«Metaclass» CallOperationAction
«Stereotype» ServiceOperationParameter
«Metaclass» OpaqueAction
«Stereotype» VariableAccess
+ numberOfVariables: Integer [1]
+ variableType: UMLPrimitiveTypes [1]
«Stereotype» FileAccess
+ filesize: Integer [1]
+ fileunit: FilesizeUnit [1]
«Stereotype» OperationAccess

Figure 6.10: Definition of Stereotypes to model services and their access

To model the frequency of a service access, the stereotype *ServiceAccessPattern* extending the *Activity* Metaclass is provided. An application of this stereotype is shown in Figure 6.11. On the left side (6.11a) the Activity *OPCUA-Server-Access* is depicted on which the stereotype ServiceAccessPattern is applied to. The Activity contains the *OperationAccess* to the actual service which is detailed in Figure 6.12. In this example, the service access has been set to a fixed interval of 1500 ms, in which two actions are executed.



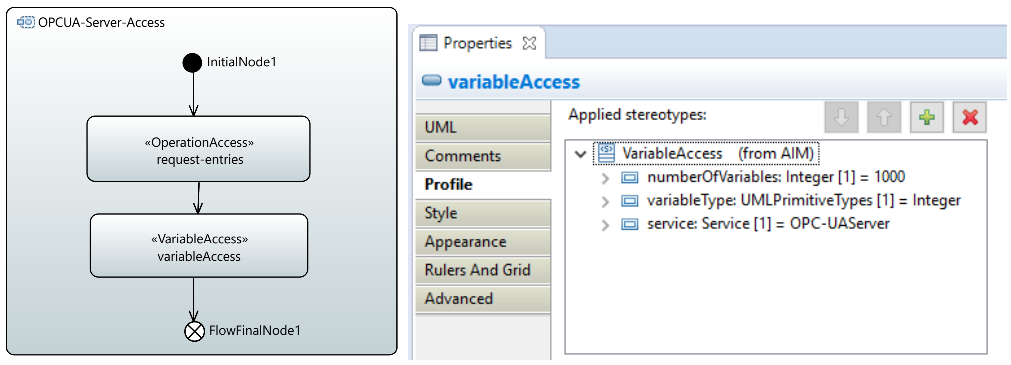
«Block, EnvironmentElementTemplate» SCADA
flowport1 : SCADA-Data
«ServiceAccessPattern» OPCUA-Server-Access
<Control Flow>
<Control Flow>
<Control Flow>
InitialNode1
«OperationAccess» request-entries
maxEntries
target
«VariableAccess» variableAccess
FlowFinalNode1
Diagram OPCUA-Server-Access
Diagram FTPServerAccess

«ServiceAccessPattern» OPCUA-Server-Access
Applied stereotypes:
ServiceAccessPattern (from AIM)
accessLiteral: String [1] = 1500
accessKind: ValueKind [1] = Fixed
timeunit: TimeUnit [1] = ms

(a) OPC-UA Server ServiceAccessPattern (b) Access properties

Figure 6.11: Modeling the *ServiceAccess* of the SCADA System Element

The Activity *OPCUA-Server-Access* can also be graphically represented with an Activity diagram. Such a diagram is shown on the left side (6.12a) of Figure 6.12. The first action is used to execute a *ServiceOperation* provided by the OPC-UA-Server. The second action requests 1000 Integer variables.



OPCUA-Server-Access
InitialNode1
«OperationAccess» request-entries
«VariableAccess» variableAccess
FlowFinalNode1

variableAccess
Applied stereotypes:
VariableAccess (from AIM)
numberOfVariables: Integer [1] = 1000
variableType: UMLPrimitiveTypes [1] = Integer
service: Service [1] = OPC-UAServer

(a) Activity with actions (b) VariableAccess properties

Figure 6.12: Definition of two actions for the *ServiceAccess*

6.1.5 IO Package

The IO package only contains stereotypes for the abstract topology independent models as shown in Figure 6.13. An example how these stereotypes can be applied is shown in Figure 6.15 the following section detailing an Interbus TSM.

The *fieldbus* stereotype is used to annotate a Class and mark it as a fieldbus. The stereotype provides additional properties like interval, type, and controller. Connected IOs are referenced via the *ioendpoints* relation to point. An OCL constraint (omitted in this diagram) checks, whether the *IOEndpoint* stereotype is applied to related parts. The IOEndpoint is an abstract Class from which IO and IOCollection are derived from. Please note, that the IOCollection slightly differs from the formal model. For a more sophisticated model-to-model transformation the number of IOs has been split into two properties (numberIOAnalog and numberIODigital). The *FieldbusFlowSpecification* element is not necessary needed to specify the TIM elements of a fieldbus. It is provided as an optional stereotype to indicate that certain *InformationFlows* are used as a fieldbus. For the TSM, such a specific flow specification is needed. Therefore, this element serves as a base class and will be extended by fieldbus specific models like the *InterbusFlowsSpecification* in the following section. The *fieldbus* property references the according Fieldbus element in the model to easily navigate to the element and simplify subsequent transformations into analysis models (see Section 6.2).

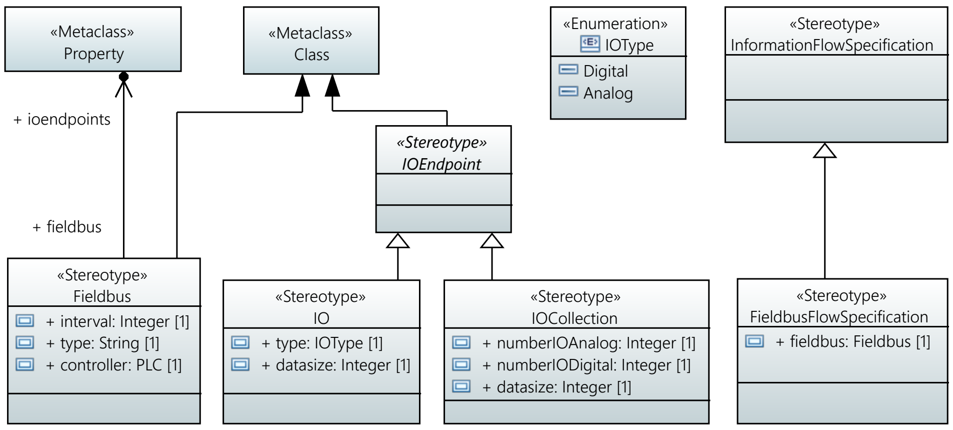


Figure 6.13: Stereotypes for the abstract IO model

Interbus TSM

The Interbus TSM is used to model Interbus specific elements in addition to the existing AIM profile. In this thesis, a new profile (aim.interbus) is created that imports the AIM stereotypes and extends them if necessary. Figure 6.14 shows an overview of the specified stereotypes of this profile. It reflects the TSM model developed in Section 5.1.5. Core element of the profile is the abstract stereotype *InterbusDevice*. It is the basis for the *RemoteBusDevice* and *LocalBusDevice*. They provide means to connect to *IOEndpoint* elements which could be either *IO* or *IOCollection*. The *Fieldbus* has been further specialized by defining an *Interbus* fieldbus element. To represent the different kinds of segments of an Interbus network (see 5.1.5), the Enumeration *InterbusConnectionType* can be used to specify the type of an *InterbusConnection*. This element extends the *FlowConnector* which is used to connect Ports of System- and EnvironmentElements. Another property of the InterbusConnection is the length of the segment. The *InterbusMaster* is used to identify the element that manages the Interbus. The OCL expression constrains the application of this stereotypes to elements that are also annotated as a *PLC*. Interbus specific flows are indicated by applying the *InterbusFlowSpecification* stereotype to flow specifications.

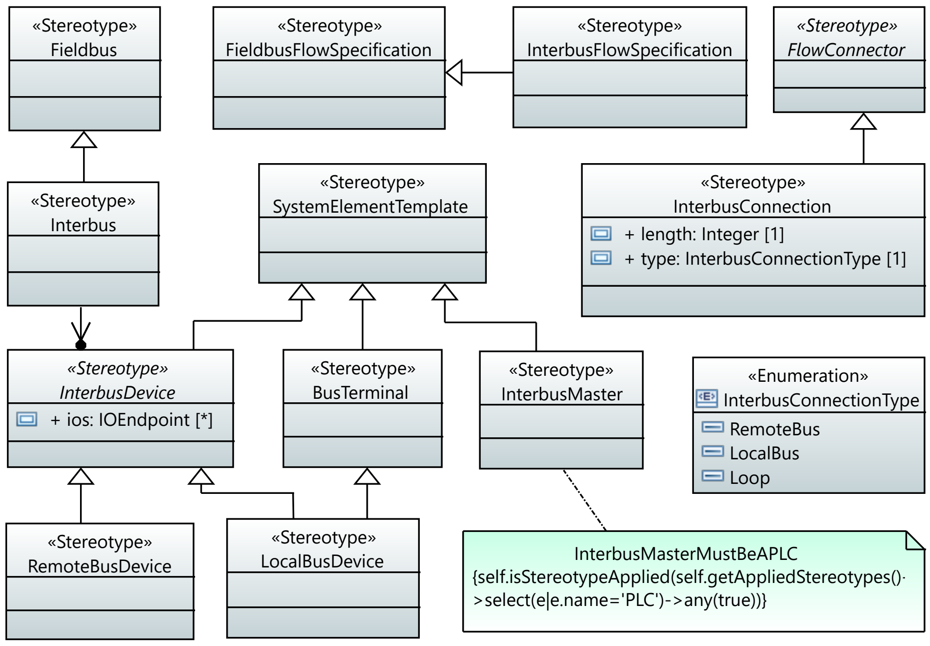

«Stereotype» Fieldbus
«Stereotype» FieldbusFlowSpecification
«Stereotype» InterbusFlowSpecification
«Stereotype» FlowConnector
«Stereotype» Interbus
«Stereotype» SystemElementTemplate
«Stereotype» InterbusConnection
+ length: Integer [1]
+ type: InterbusConnectionType [1]
«Stereotype» InterbusDevice
+ ios: IOEndpoint [*]
«Stereotype» BusTerminal
«Stereotype» InterbusMaster
«Enumeration» InterbusConnectionType
RemoteBus
LocalBus
Loop
«Stereotype» RemoteBusDevice
«Stereotype» LocalBusDevice
InterbusMasterMustBeAPLC
{self.isStereotypeApplied(self.getAppliedStereotypes()->select(e|e.name='PLC')->any(true))}

Figure 6.14: Stereotypes for the Interbus TSM model

An application of the general IO (Figure 6.13) and the topology specific Interbus stereotypes (Figure 6.14) is shown in Figure 6.15. Depicted is an excerpt of the Turbocharger example introduced in Chapter 3. All SysML4CONSENS specific stereotypes like *SystemElementExemplar* or *InformationFlowSpecification* (see Section 2.2.3) are neglected in this figure for a better overview. The *PLC* and *InterbusMaster* stereotypes have been applied to the *Unit2PLC*. The *PLC* stereotype provides additional attributes like vendor and type that are omitted in this figure (see Screenshot 6.5). A *Unit2Fieldbus* element represents the Fieldbus and the *Interbus* marks it as an Interbus element. All ports are typed by *InformationFlowSpecifications* and are further detailed by applying the *InterbusFlowSpecification* and *FieldbusFlowsSpecification* from the AIM base profile. To improve visibility of the figure, only the ports of the PLC and the Unit2Fieldbus are indicated with such stereotypes. The PLC is connected via the Interbus fieldbus to the AssemblyConnector. This element contains a *BusCoupler* and two actuators *FixingL2Right* and *FixingL2Left*. The actuators are annotated by the *IO* stereotype and provide additional information of type and datasize. The BusCoupler is a *RemoteBusDevice* and holds a list of connected sensors and actuators via the IOs property. The connections between

BusCoupler and the actuators are also annotated via the *InterbusConnection*, but neglected in this figure to improve the visibility.

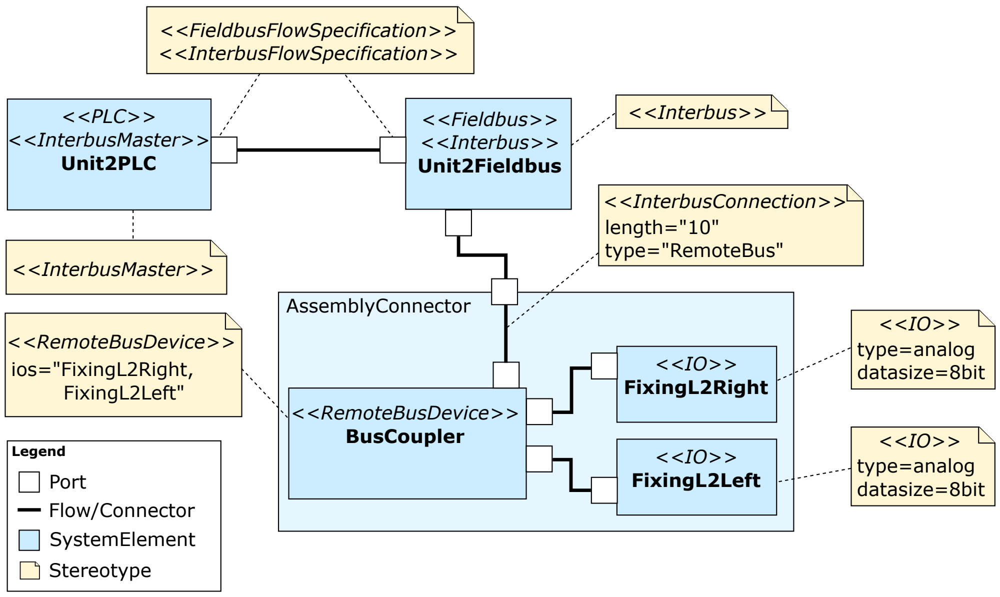


Figure 6.15: Exemplary application of Interbus stereotypes

6.2 Performance Analysis with Palladio

In the second part of this Chapter, approaches and tools for a performance prediction of an automation system are evaluated. As a result of this evaluation, the Palladio Modeling Framework has been selected for this thesis. It provides means to performance model, simulate, and analyze an automation system. The models necessary for the simulation of the different influence factors and their parameters are further detailed in the following subsections. Palladio is used to simulate a Phoenix Contact ILC 171 ETH 2TX with a selection of fieldbusses and services.

6.2.1 Requirements

To select a fitting performance prediction approach or tool, the following requirements have been defined. In section 6.2.2 each tool will be checked to which degree the requirements are satisfied.

- **Req1** *Holistic system:* The tool must be able to analyze complete automation systems. This includes networks as well as internal hardware or software of the PLC. The simulation must provide at least the overall utilization of the CPU(s).

- **Req2** *Hardware dependent and independent modeling:* The tool must be able to model and analyze specific execution times as well as hardware independent resource usages/demands.
- **Req3** *Modeling effort:* The tool must be easy to use and understandable for experts of the domain automation system allowing them to create or update influence factors for a PLC.
- **Req4** *Toolchain:* The tool must be easy to integrate into the selected toolchain. This means, that the annotated CONSENS modeled created with Papyrus should be easily transferred to the selected analysis tool.
- **Req5** *Analyzability of the performance prediction results:* The tool must provide adequate means to analyze the prediction results. This includes foremost the CPU utilization followed by other details like execution times of different influence factors or processes.
- **Req6** *Extensibility:* The tool must offer the possibility to extend it. Therefore, the code or adequate extension mechanisms should be provided. This will help to add new features like custom made schedulers.
- **Req7** *Embedded systems:* The tool must be applicable for the automation domain, respectively focus on embedded systems. This means it must provide the capability to model behavior and structure in a sufficiently detailed way to perform performance predictions in the early stages of the development.

6.2.2 Related Work: Performance Prediction Approaches

To model and analyze the performance of a single PLC, a whole automation system, or just the used fieldbus, several tools with varying approaches can be used. Each approach has a different focus and underlying techniques to provide the necessary evaluation data. They differ mostly in their focus and application domain, spanning from embedded [Hap05, Wan06], non-embedded [WS02], distributed [TP09, CHL$^+$03, LWF08, FCF$^+$13], with fieldbus [LF07, LF12, COH07, MDFF06b, HCÅ03] to just standalone PLCs [FH12, FHMB13]. In [Per06] an evaluation and comparison of performance analysis methods for distributed embedded systems are given. Another survey compares different model-based performance prediction approaches [BdMIS04]. This thesis focuses on the application domain of automation systems. The goal is to find a suitable analysis tool for the early validation with regard to a high level of abstraction and to develop an automatic generation of input models for it.

As mentioned, exist a growing number of performance prediction approaches for the area of embedded and distributed embedded systems. The following approaches have been further investigated and their applicability towards this thesis goal and identified requirements checked.

- **A1** *TrueTime:* TrueTime [CHL+03, HCÅ03, COH07] is a MATLAB [Mat16] and Simulink [Mat17] based simulator for real-time control systems. The framework allows the specification, programming, and simulation of programs, threads, real-time kernels, schedulers, network transmissions, and continuous plant dynamics. The toolbox for MATLAB/Simulink provides blocks that can be easily parameterized and further extended. Such a block can abstract a complete fieldbus used to transport control commands and data. Supported busses are, among others, Ethernet, FlexRay, and PROFINET. To simulate Programs or FB, TrueTime executes actual code written either as C ++ functions or as MATLAB M-files. It is possible to extend existing blocks and create additional ones. To conduct a performance analysis, the system under development must be programmed in detail. By using the MATLAB standard functions to gather and visualize (performance) data, TrueTime provides detailed simulation results. For the domain of automation systems, MATLAB provides a well-known programming system. However, creating the simulation models as code, the TrueTime blocks, and the hardware independent resource usages can be quite difficult. Additionally, is an easy integration into the toolchain complex. More information on TrueTime can be found in their extensive manual [CHO10].
- **A2** *Palladio:* The Palladio Component Model (PCM) [BKR09, RBB+11] is an architecture description language supporting performance evaluations of component-based software systems. These software systems cover distributed processing resources which can be connected via networks. The framework provides several ways to analyze the modeled systems like Simucom. Simucom generates a simulation based on layered queueing networks. Simucom also provides fine grained sensor data of the simulation, allowing the performance analyst to conduct in-depth investigations of the system under development. Palladio is open source and based on the Eclipse Modeling Framework (EMF) technology [SBMP08, Ecl17b] which allows an easy generation of input models. Via extension mechanisms, sensors and custom operating system schedulers can be created and added to the Simcom simulation. Palladio is realized with the Java programming language. Other extensions for Palladio can be used to incorporate message based systems or to generate input models for other simulation approaches like OMNeT++. However, Palladio is designed to predict the performance of server and desktop systems. Its applicability to simulate embedded systems must be ensured or realized.
- **A3** *OMNeT++:* The Objective Modular Network Testbed in C++ [Ope17, VH08] is a modular, component-based C++ simulation library and framework. The focus of this framework is modeling and simulation of complex networks. Vital part of the framework is the ability to create custom modules for new types of networks or to modify existing ones. OMNeT++ also provides a rich set of utilities (makefile creation tool, etc.) which make the creation of various extensions possible [SKKS11, V+01]. They enable the system analysts to simulate large scale IP networks as well

as small, detailed, embedded wireless networks. The Eclipse based IDE (not EMF-based) can be used to program modules and compose them with the high-level language (NED). Still, OMNeT++ requires extensive knowledge to create new modules for the simulation like schedulers, networks, and services. Palladio provides an extension called OMPCM [HMR13] which generated detailed OMNeT++ simulation models from PCM models. It uses a specialized representation for description of RD-SEFF behavior called SimCore. This extension allows a more accurate analysis of the network and its properties but not an in-depth analysis of the PLCs performance. Therefore, OMNeT++ can be used either standalone with its own models and simulation engine or seen as one of the many ways Palladio can execute a performance analysis. However, for the selection of a fitting prediction approach, OMNeT++ will be regarded as the standalone version.

- **A4** *TimingArchitects:* The Timing Architects Tool Suite [TA17] provides a set of tools to model, simulate, and analyze embedded devices for the goal of optimizing signal and execution chains. The TA tool suite can import AUTOSAR [AUT17] System Description and/or ECU Configuration files as well as Amalthea [AMA17] models. The latter are based on EMF technology, which could be used to generate input models and integrate the tool into the desired tool chain more easily. The Amalthea models allow the definition of AUTOSAR runnables, tasks, buses, and more design artifacts. These runnables are specified via hardware independent instruction sets. The hardware is modeled separately with the capability to execute a certain amount of instructions per time unit. The TimingArchitects tool suite is a commercial software and does not provide means to incorporate custom made scheduler. However, it allows in-depth definition and analysis of call graphs and timing aspects including the distribution of metrics like response time.
- **A5** *Real-Time Calculus (RTC) Toolbox:* This approach is also based on MATLAB and provides a tool box named Real-Time Calculus (RTC) Toolbox [Wan06]. RTC provides libraries to perform a modular, interface-based design and a performance analysis based on variability characterization curves (VCCs) [MZCW04]. For this, the approach analyzes the flow of event streams over resources to derive performance characteristics of the modeled system. To analyze a distributed embedded system, the system analyst is required to write a MATLAB program. This program invokes commands for the creation and analysis of performance networks. As mentioned in [Per06], will this process take considerable modeling effort for larger systems. The toolbox provides means to specify workloads, streams, arrival times, resource components including scheduler and scheduling strategies on a low level. Therefore, high-level modeling of field-busses, communication networks, tasks, programs and so on must first be mapped onto these elements. The input model is described as a MATLAB model in combination with M-code, the MATLAB programming language.

- **A6** *ModelicaNCLibrary:* Frey et al. provide a modeling and simulation framework [LWF08, FL09] in form of two libraries for Modelica [Mod17b]. The first library provides models for Ethernet, WLAN and ZigBee networks, whereas the second library contains models for different controller types and interfacing devices. Their approach can be used to predict the response times of complex automation systems with control loops based on different network types and topologies [LF07, LF12, MDFF06b]. It supports tasks, processes, different scheduler and scheduling strategies. Additionally, it can be used to model dedicated hardware like an analog-to-digital converter. The Modelica models must be coded in either the Modelica language or in external functions in C or Java. The libraries are provided as open source software which allows to extended or to create a custom scheduler. The Modelica models are text based and can be generated. Also, Modelica provides a rich set of tools to visualize different parts of the simulation, further supporting the analysis.

- **A7** *chronSIM:* The Inchron tool suite [INC17] can be used to simulate, visualize, and analyze design alternatives during the development of an embedded system. Part of the tool suite is chronSIM [AADG12] which can be used to simulate the modeled system. Input for a simulation are C files for the AUTOSAR runnables as well as a project description files that further defines settings like tasks. These files can be generated by external tools. With further extensions is chronSIM able to incorporate various bus systems as well as different operating system scheduler. Additional tools in the suite allow an in-depth analysis of task executions and message travel times. Like Timing-Architects, is the tool focused on the automotive domain.

Based on the previously defined requirement and introduced approaches, the following Table 6.1 could be defined. Each tool or approach needs to be slightly modified to incorporate the desired changes like a custom scheduler. Also, is requirement 2 (hardware independent and dependent modeling) not fully supported by one approach. However, this requirement can easily be covered by generating hardware dependent models for each new simulation. Best candidates for a simulation based analysis are TrueTime, Palladio, and the Modelica-based ModelicaNCLibrary. For this thesis, Palladio has been selected due to its coverage of most requirements, the component-based modeling approach, the various extensions available, it's easy to extend framework, and its already high number of different underlying simulation engines (which also include OMNeT++). In [dGJKK12] a similar search and selection has been made, leading to Palladio as a fitting approach.

6.2.3 Palladio Simulation Models

This subsection briefly explains the Palladio models that will be used to predict the utilization of a PLC. For this purpose, the influence factors, parameters, and additional information are extracted from the Systems Engineering models

Table 6.1: Overview of requirements and approaches

	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7
TrueTime	+	o	-	-	+	+	+
Palladio	+	o	+	+	+	+	o
OMNeT++	+	o	o	o	o	o	+
TimingArchitects	o	o	o	o	+	-	+
RTC Toolbox	-	o	-	-	-	o	+
ModelicaNCLibrary	+	o	o	o	+	+	+
chronSIM	o	o	-	o	+	-	+

Legend

+ satisfies requirement
0 partially satisfies requirement
- not capable

and based on this different Palladio models, their elements, and relations are generated. The following sections will briefly introduce the decisions why and how the elements are created. The full model containing all influence factors and their PLC specific performance profiles and resource demands will not be shown. This is to protect the intellectual property, internal structure, and properties of the firmware that Phoenix Contact provided as a basis for an industry driven project which is used as a basis for this thesis.

Palladio uses five input models to specify the system and its environment as shown in Figure 6.16. The *ResourceEnvironment* model is used to specify the hardware, networks, and connections. The structure and behavior of software components is defined in the *Repository* model. Core parts of this model are *BasicComponents* and *Interfaces*. Similar to Java, define the *Interfaces* a set of operations to execute and the *BasicComponents* implement them. Vital part of the *BasicComponent* are Service Effect Specifications (SEFF) that describe the behavior of each service. They model the order and extent of resource usage (e.g. CPU resource demand) as well as calls to other (external) components. The *System* model is used to compose the components and create instances for the simulation. To specify which software component is run on which hardware, the *Allocation* model is used. Finally, the *UsageModel* is needed to specify a scenario that describes the usage intensity by indirectly invoking the provided SEFFs. More details to the five Palladio models can be found in the Foundations Chapter 2. A sixth model that is used for the simulation of an automation system is a an additional task model necessary to provide information to the custom scheduler. It is not provided by Palladio. Details about this model are given in the PLC section.

The following subsections detail the realization of selected influence factors and their parameters in Palladio models. The focus remains on the Repository and UsageModel which are used to specify most of the structure and behavior of the system. The other models are necessary but mostly omitted here. This is possible, because the Allocation model only provides a single hardware. Therefore, all component instances will be deployed on one target. The component instances reflect the created components usually in a 1:1 relations, meaning that each component will be instantiated only once. Only exception from this rule

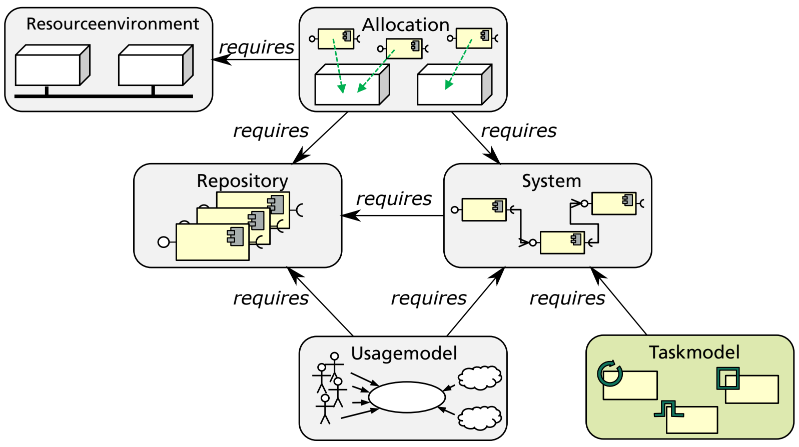


Figure 6.16: Overview of the used model types

are Function Blocks, which can be instantiated more than once.

Model elements overview:
The following Table 6.2 shows a simplified mapping of influence factors to Palladio model elements. In the first two columns influence factors and their parameter are listed. All following columns indicate which elements in the different simulation models need to be created. The influence factor PLC contains several parameters which are mapped differently. CPU, Cores, and Scheduler are solely mapped to the Resource model via *ResourceContainer*. Operating System (OS), IPTraffic, and Firmware put a load onto the CPU based on specific access patterns. Therefore, *Interfaces*, *BasicComponents*, *SEFFs*, and resource *Demands* need to be created. They are triggered by their respective *workloads* (open or closed) in the Usage Model. To set up task priorities for the scheduler, the Task model is used including different *TaskInfo* elements. Instances of the BasicComponent are composed to a system in the System Model. This is done by generating *AssemblyContexts* and corresponding *Connectors* and *Ports* to assemble the system. Finally, the Allocation model is used to map the AssemblyContexts onto the ResourceContainer. The last two steps are necessary for each influence factor that generates Interfaces and BasicComponents.

The generated Palladio elements in the different models are briefly explained in the following subsections. Similar steps that needs to be performed, like the creation of the AssemblyContexts and the mapping onto the ResourceContainer, are omitted. A field containing a *(>task)* symbol indicates that this element is triggered by a corresponding task Workload element. The Program, for example, is only executed if the task is triggered in the usage scenario.

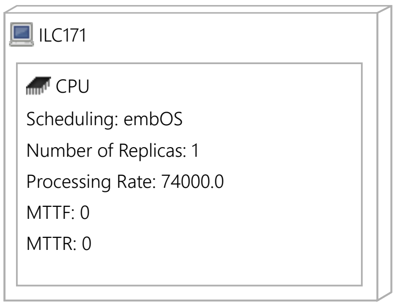
PLC

The Palladio resource environment model contains all elements to specify execution environments and their connections to each other. The resource container

		Repository				Usage	Resource		Taskm.	System		Allocation
Faktor	Parameter	Interface	BasicComponent	SEFF	Demand	Workload	Res.Container	ProcessingRessource	TaskInfo	AssemblyC.	Connector + Port	AllocationContext
PLC	CPU	x	x				x	x				
	Cores							x				
	Scheduler						x	x				
	OS	x	x	x	x	x			x	x	x	x
	IPTraffic	x	x	x	x	x			x	x	x	x
	Firmware	x	x	x	x	x			x	x	x	x
Program	Exec.Time				x							
	AccesFreq.					(>task)						
	Core Aff.							x				
Function Block	BaseLoad	x	x	x	x					x	x	x
	AccesFreq.					(>task)						
	Operations	x	x	x								
	Load/Op.				x							
Function	AccesFreq.	x	x	x		(>task)						
	Load/Op.				x							
Task	Priority	x	x	x		x			x	x	x	x
CyclicTask	CycleTime	x	x	x		x			x	x	x	x
EventTask	AccesFreq.	x	x	x		x			x	x	x	x
IdleTask		x	x	x		x			x	x	x	x
IO	(all param.)	x	x	x	x	x			x	x	x	x
Services	(all param.)	x	x	x	x	x			x	x	x	x

Table 6.2: Generated Palladio elements for each influence factor

shown in Figure 6.17 is used to model the PLC for the Unit 2. The container itself contains a processing resource which represents the CPU of the ILC 171 ETH 2TX with a processing rate of 64.000 resource units per millisecond, corresponding to a CPU with 64 MHz. This processing rate is the base value for all calculations that transform an *ExecutionTimeSpecification* into a Palladio compliant resource demand.


ILC171

CPU
Scheduling: embOS
Number of Replicas: 1
Processing Rate: 74000.0
MTTF: 0
MTTR: 0

Figure 6.17: Resource container for the ILC 171 PLC of Unit 2

An important property of the processing resource is the selected scheduling. Palladio provides basic scheduling algorithms and a framework to specify custom scheduler (see [Hap08, Hap04, Hap16]). Using this framework, a developer is able to create a scheduler with specific settings. These settings include quantum time slices, queuing configurations, starvation boost and priority settings. The ILC 171 uses *embOS* as an underlying operating system. It is available as source code, allowing the Phoenix Contact firmware developers to access in-depth details of the specification and make modifications to the scheduler. To support the simulation of an *embOS* based PLC, the details of the scheduler had to be incorporated into Palladio.

In addition to the configuration, the behavior of the Idle-Task (or Default-Task for Phoenix Contact PLCs) must be implemented. The Idle-Task is executed in a loop with a very specific waiting time between executions. These waiting times are calculated based upon the last loop duration. Figure 6.18 visualizes this calculation. Each Idletask period can be split into the actual execution of the task and a waiting time. If the idle task is preempted by higher priority tasks, this time is also taken into account and added to the duration. Afterward, the wait time is calculated and the Idletask execution is postponed so that all lower priority processes can be executed in this time frame.

The Palladio Metamodel lacked the ability to specify scheduler priorities for different *BasicComponents*. To create a custom made scheduler for simulating the embOS IdleTask, a sixth *Taskmodel* has been developed and added to the necessary input models as shown in Figure 6.16. The *Taskmodel* simply provides priorities and task type definitions and relates them to elements in the *System* model. During the simulation, the scheduler reads the model and internally manages task priorities and behavior accordingly.

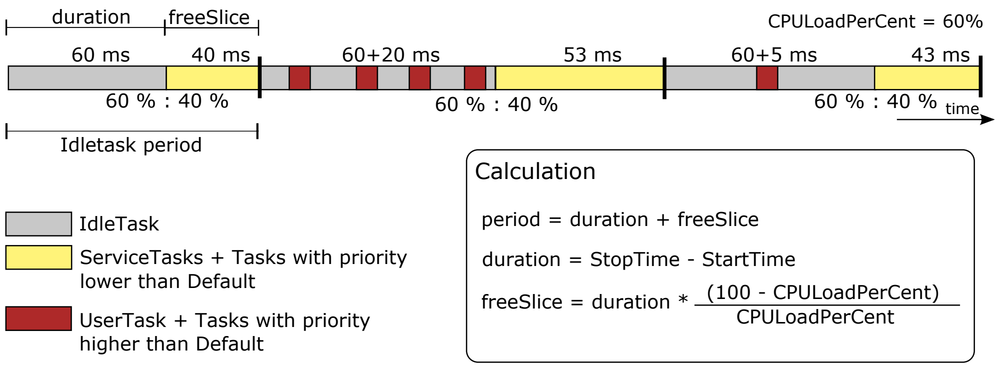

duration
freeSlice
CPULoadPerCent = 60%
60 ms
40 ms
60+20 ms
53 ms
60+5 ms
43 ms
60 % : 40 %
60 % : 40 %
60 % : 40 %
time
Idletask period
IdleTask
ServiceTasks + Tasks with priority lower than Default
UserTask + Tasks with priority higher than Default
Calculation
period = duration + freeSlice
duration = StopTime - StartTime
freeSlice = duration * (100 - CPULoadPerCent) / CPULoadPerCent

Figure 6.18: Specifics of the embOS Scheduler

OS, Firmware and IPTraffic

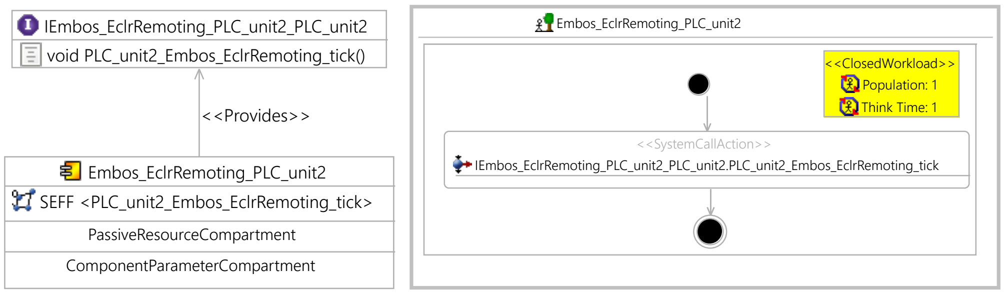
Each PLC has a set of processes and threads that will form a unique load profile. For the ILC 171 ETH 2TX, different firmware and operating system processes must be considered when creating Palladio simulation models. To do this, the details of the PLC must be investigated. Fortunately, offers a special debugging firmware for the ILC 171 a way to conduct in-depth profiling of the hardware and software. More information on this profiling mechanism can be found in Chapter 7. To create a load profile for the ILC 171, the most influential processes and threads have been identified under different load conditions. These processes have been condensed and combined into three standalone demands that will put a load onto the CPU.

- The **embOS baseload** represents the background utilzation induced by the underlying operating system
- The **SystemTick** is a timer integrated into the operating system. It is used to manage or trigger all PLC specific functions or processes and invoked every millisecond.
- The IEC Runtime-Environment **eCLR** provides a rich set of functions that can be used by developers. These functions usually put a load onto the CPU when invoked, but some of them also run continuously in the background. To exchange (engineering) data with the PLC, a vendor specific communication stack has been implemented. This stack is called **Remoting** and is a major part of the eCLRs background utilization.

For each of these three firmware specific loads, a Palladio *BasicComponent* and according *Interface* has been created. These components are instantiated in the System model and called from dedicated UsageScenarios. Figure 6.19 shows the BasicComponent in the Repository model (6.19a) and the UsageScenario (6.19b) for the EclrRemoting baseload. The SEFF of the BasicComponent contains a ResourceDemand specified as

$$DoublePDF[(8640.0; 0.0)(9120.0; 0.05)(10080.0; 0.9)(10560.0; 0.05)]$$

to emulate a jittering of the load and not just a fixed value.


IEmbos_EclrRemoting_PLC_unit2_PLC_unit2
void PLC_unit2_Embos_EclrRemoting_tick()
<<Provides>>
Embos_EclrRemoting_PLC_unit2
SEFF <PLC_unit2_Embos_EclrRemoting_tick>
PassiveResourceCompartment
ComponentParameterCompartment
Embos_EclrRemoting_PLC_unit2
<<ClosedWorkload>>
Population: 1
Think Time: 1
<<SystemCallAction>>
IEmbos_EclrRemoting_PLC_unit2_PLC_unit2.PLC_unit2_Embos_EclrRemoting_tick

(a) Repository component (b) UsageScenario

Figure 6.19: Component and UsageScenario for the eCLRRemoting load

This same modeling approach is used to model the general IPTraffic which must be handled by the PLC. The parameters, resource demand, and UsageScenario arrival times can be derived from the properties defined by the AIM profile.

Programs

Each Program is part of a Task which triggers it. A Task can contain multiple Programs, but at least one. To model Programs in Palladio, two different approaches can be used. First, the Program will be represented by a *BasicComponent* and it's according *Interface* like previously shown. This option allows a much better reuse of the created model elements. However, due to the fact that the majority of all Palladio models are generated and the Tasks already need a *BasicComponent* and *Interface*, the Program specific actions and loads can as well be added to this component. This solution simplifies the models and decreases the time to simulate it.

Therefore, each Program is part of a Task (see subsection 6.2.3) with an associated *SEFF*. The *SEFF* is used to specify, for example, probabilities for executions, parameters, or *InternalActions*. The *InternalActions* provide the ability to set a resource demand for a CPU. Figure 6.20 shows three different *InternalActions* executed by two SEFFS (MainTask and MillTask). The actions are named after the Program name. The *ResourceDemand* is set to a specific demand, derived from the properties of the AIM profile. In this particular case, the WCET (or fixed time) has been calculated by the resource units the CPU can process. Other value kinds like Bounded, RandomSet, and RandomInterval can be mapped to Palladio by using the Stochastic Expression EBNF specified by Palladio. If a task executes multiple Programs as shown in the example, the different Program InternalActions are added to the SEFF and executed in order.

Functions and Function Blocks

To simulate Function Blocks and Functions in Palladio, all *Interfaces* and *BasicComponents* need to be created first. Figure 6.21 shows on the left side

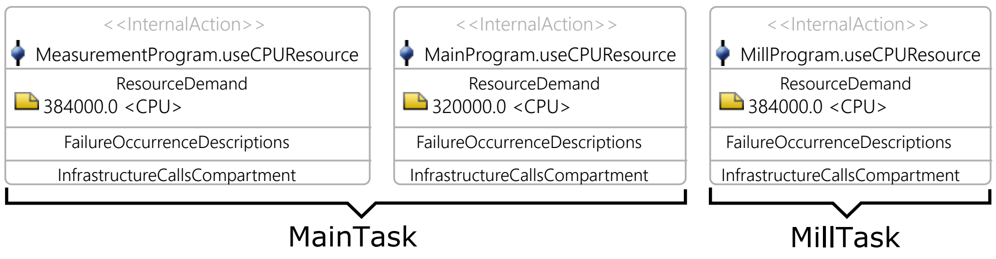
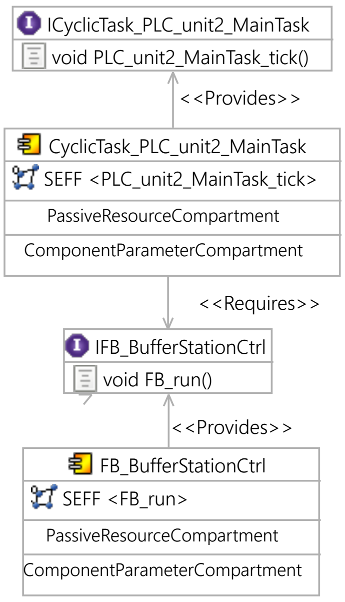
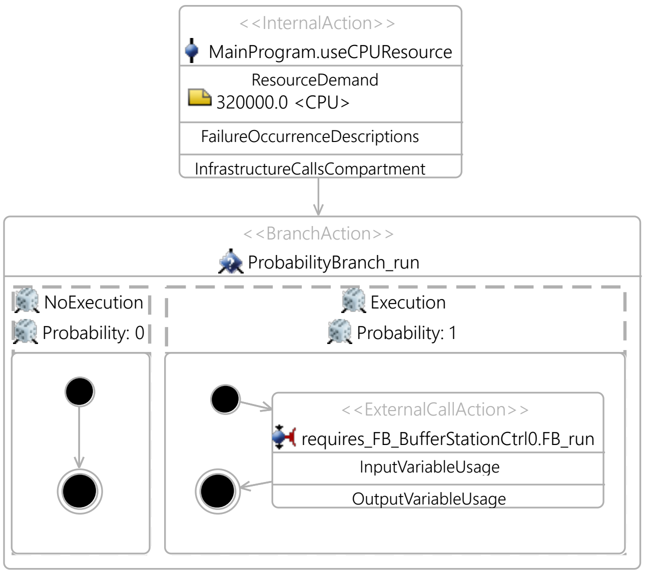

<<InternalAction>>
MeasurementProgram.useCPUResource
ResourceDemand
384000.0 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment
<<InternalAction>>
MainProgram.useCPUResource
ResourceDemand
320000.0 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment
<<InternalAction>>
MillProgram.useCPUResource
ResourceDemand
384000.0 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment
MainTask
MillTask

Figure 6.20: Realization of Programs with different kinds of Actions

(6.21a) an excerpt from the Palladio Repository model. The figure shows a *BasicComponent* representing the *CyclicTask* and the included Programs. The *BasicComponent* does not only provide its according Interface but also requires the Interface of a used Function Block (*IFB_BufferStationCtrl*). The operations of this *Interfaces* can be called from inside the CyclicTasks SEFF as shown on the right side (6.21b). After the InternalAction for the Program has been executed an ExternalCallAction is performed. This ExternalCallActions triggers the execution of the BufferStationControl's *FB_run* operation and its contained *SEFF* and *ResourceDemand*. This approach can be repeated to model nested call hierarchies, including Function Blocks to Function Block calls.


ICyclicTask_PLC_unit2_MainTask
void PLC_unit2_MainTask_tick()
<<Provides>>
CyclicTask_PLC_unit2_MainTask
SEFF <PLC_unit2_MainTask_tick>
PassiveResourceCompartment
ComponentParameterCompartment
<<Requires>>
IFB_BufferStationCtrl
void FB_run()
<<Provides>>
FB_BufferStationCtrl
SEFF <FB_run>
PassiveResourceCompartment
ComponentParameterCompartment

(a) Repository



(b) SEFF

Figure 6.21: Repository components and SEFF for calling Function Blocks

To model PLC specific Functions, the same approach can be used. First, a global *BasicComponent* and *Interface* are created for a specific PLC. Then all Functions from the AIM profile are gathered and added to the Interface. This allows other *BasicComponents* to use the provided Operations.

Tasks

Tasks are used to trigger the execution of Programs and Function Blocks. Therefore, they are mapped directly to UsageScenarios. Figure 6.22 shows the specification of the *MillTask* and *MainTask*. The MainTask is the cyclic Task that executes all associated Programs every 150ms. Therefore, a UsageScenario has been set up using an OpenWorkload with an InterarrivalTime of 150. The SystemCallAction will run the according Program SEFFs and their InternalActions as defined in 6.2.3.

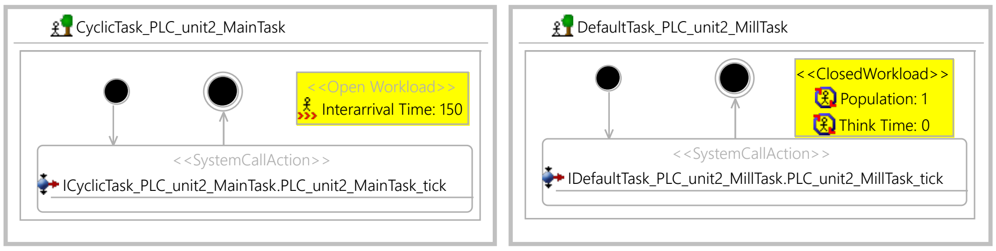

CyclicTask_PLC_unit2_MainTask
<<Open Workload>>
Interarrival Time: 150
<<SystemCallAction>>
ICyclicTask_PLC_unit2_MainTask.PLC_unit2_MainTask_tick
DefaultTask_PLC_unit2_MillTask
<<ClosedWorkload>>
Population: 1
Think Time: 0
<<SystemCallAction>>
IDefaultTask_PLC_unit2_MillTask.PLC_unit2_MillTask_tick

Figure 6.22: UsageScenarios representing the *MillTask* and *MainTask*

The second UsageScenario shown in the figure represents an IdleTask (named DefaultTask for ILC 171). It uses a ClosedWorkload instead of an OpenWorkload due to its reoccurring (looping) nature. The *ThinkTime* property is not used for the IdleTask. The custom made embOS scheduler, introduced in 6.2.3, calculates the ThinkTimes dynamically based on previous execution times and defined system settings.

Services

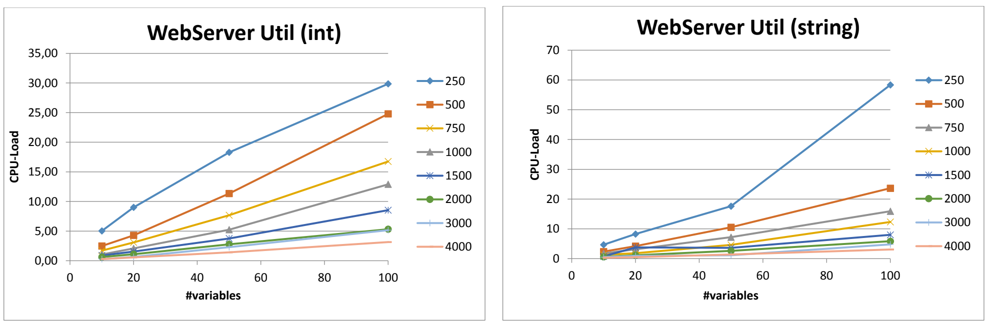
Services are realized the same way Function Blocks are created. For each Service, a Palladio BasicComponent and its according Interface is created. Via the provided Operations, the SEFFs and their resource demands are executed. The SEFFs are derived from the AIM profile properties and all similar loads are combined, respectively accumulated into one access. For example, if multiple accesses are executed by the same *ServiceAccessPattern*, then these will be combined into a single operation with its according UsageScenario. The same applies to subsequent *VariableAccess* calls. Of course, this does not apply, if the calls are executed with different *ServiceAccessPatterns*. Services currently realized are the OPC-Server, Webserver, and FTPServer. For each Server/Service dedicated measurements have been made using the profiling mechanism of the special debugging firmware provided for the ILC 171 PLC. These measurements are the basis for creating service specific load profiles. Some details about these measurements can be found in Chapter 7.

In the following subsection, an example for the creation of a Webserver load profile will be given. The Webserver is a core feature of the ILC 171 ETH 2TX

PLC that is used to provide a web-based HMI for application specific data. With a special tool, included in the PC Worx Engineering tool suite called WebVisit, the available global variables can be selected and visualized on an HTML page with Java script. The Webserver on the PLC reads or writes the data in fixed intervals set by the WebVisit tool.

To check the different parameters of the Webserver influence factor, several varying WebVisit setups have been made, downloaded onto the PLC and executed. Afterward, the profiling mechanism provided detailed information about the CPU load induced by the Webserver. The influence factor has three basic parameters: The number of variables accessed from the Webserver, the kind of variable (e.g. String or Integer) and the set refresh interval.

Figure 6.23 shows two graphs visualizing the results of multiple measurements using Integer and String variables, varying refresh times, and number of variables. All graphs show the CPU-Utilization on the y-axis as a percentage value and the number of variables is given on the x-axis. Each line represents the results of a measurement with refresh intervals set to 250 ms, 500 ms, 750 ms, 1000 ms, 1500 ms, 2000 ms, 3000 ms, and 4000 ms. On the left side (6.23a), the measurements for the variable type Integer and on the right side (6.23b) type String are shown. As expected is the utilization of the CPU highest for short intervals and increasing number of variables. At a refresh rate of 250 ms and 100 variables, the difference in CPU utilization between Integer (29,86%) and String (58,34%) is almost 28,48 percentage points. Based on the measurements, formulas have been developed that are used to calculate ResourceDemands for the SEFFs during the generation of the Palladio models.



(a) Variable type Integer (b) Variable type String

Figure 6.23: Visualization of the Webserver utilization

IO - Interbus

To simulate the utilization induced by the Interbus fieldbus system, a BasicComponent, Interface, and associated SEFFs are created. The *ResourceDemand* is also calculated based on the properties specified by the AIM profile. The ILC 171 ETH 2TX uses a special hardware to fulfill the function of the Interbus

Master. Therefore, a major part of the workload must not be processed by the CPU. However, several threads are used to gather and copy the data to the hardware. To create a load profile for the Interbus fieldbus, the performance of the Interbus Master process has been profiled.

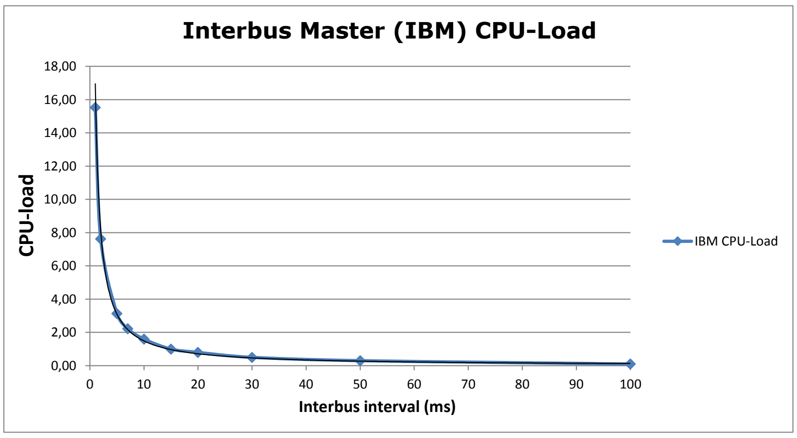


Figure 6.24: Visualization of the Interbus Master utilization

The results of these measurements are shown in Figure 6.24. The utilization has a peak of 15,6% at a fixed cycle time of 1ms and is decreasing rapidly at higher cycle times. The measurements were performed using a single Interbus device and a data size of eight Bit. For this setup, the following formula has been derived which is used to calculate the ResourceDemand.

$$f_{ibm}(i) = 16,95 * i^{(-1,065)}$$

The SEFF is executed via an UsageScenario with an OpenWorkload and a fixed InterArrivalTime of 1. Therefore, the calculated background workload is put evenly on the CPU.

The load profile for the PROFINET fieldbus has been created in a similar way. The measurements for PROFINET have been provided by Phoenix Contact for a different PLC. They are much more detailed and therefore, allow a fine-grained analysis and subsequent specification of a formula.

6.3 Overview of Models and Transformations

The goal of this thesis is to model selected influence factors in the early system development phases and perform an performance analysis. CONSENS has been selected as one of multiple approaches to create Systems Engineering models. These models are extended by automation system specific information.

To conduct a performance analysis, the Palladio Component Framework has been selected. Transforming the SysML-based CONSENS models directly into

Palladio input models is a valid step. However, the use of multiple models in a chain of smaller transformations provides several advantages. First, the CONSENS models include far more information than used for the Palladio simulation, making a single transformation more complex. An example is traversing different information flows from source to target over several hierarchies. Therefore, only the relevant information should be selected and transformed in a first step. Second, the performance prediction approach considers different PLCs from various vendors. Each PLC provides their own specific parameters contained in load profiles which are not governed in the CONSENS respectively influence model. Additionally, some parts like task management or Function Block creation in the Palladio models are interchangeable between PLCs and vendors. Therefore, it is possible to identify modules which can be reused between PLC products or product families as well as different vendors. Another advantage of using multiple models is the aggregation of information. It is, for example, necessary to aggregate all variable accesses to the OPC-Server for a primitive type like String or Integer. This can be easily done in an intermediate model. And finally, a more technical reason for using multiple steps towards the final simulation models is the handling with dedicated, PLC specific models that are much easier to parse, debug, and create transformations for.

Figure 6.25 shows the chain of models and transformation steps from the extended CONSENS model on the left side up to the final simulation results on the right side.

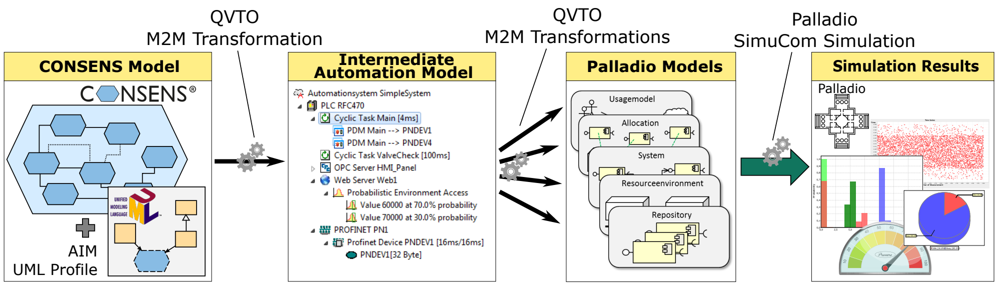


Figure 6.25: Chain of models and transformations

Creating performance models with Palladio is a complex and time-consuming task. For this reason, the intermediate *Automationmodel* has been created. It hides the Palladio models from the developer and further adds information for the simulation engine. Between the CONSENS and Automationmodel, as well as the Automationmodel and the five Palladio models (repository, usage, allocation, system, and resourceenvironment), automated transformations can be used. The transformations are performed with QVT-Operational (QVTO)[Obj11b] (see Section 6.3.2).

This section first briefly introduces this intermediate *Automationmodel* that reduces the overall complexity of the required transformations and also simplifies the debugging/development. Afterward, the set of transformations used to generate the Palladio models from the Automationmodel will be detailed. To complete the chain of models, the following section will sketch the transformations used to generate the Automationmodel from the CONSENS respectively CONSENS4SysML Papyrus models.

Similar to the Palladio models, contain most of the model-to-model transformations confidential information about the Phoenix Contact project. Therefore, only selected parts are shown in the following subsections and the appendix. They are sufficient to explain the basic properties and functionality of this approach.

6.3.1 Automationmodel

The intermediate model called *Automationmodel* is used to bring modularity (up to a certain degree) into the transformation process. Instead of performing one large transformation, two smaller and simpler transformations can be used to generate the target Palladio models. The Automationmodel is only a subset of the original Systems Engineering model and focuses only on the selected PLC that will be simulated. It combines multiple influence factors and condenses the information, making the transformation to Palladio models easier. As mentioned in Chapter 1 are the simulation model and the automation model a result of an industry project conducted for Phoenix Contact. Therefore, details about the model elements and their properties can not be given. However, the coarse structure of the model and examples of transformations creating or using it are presented. Since the Automationmodel is only used as an intermediate step to create the final output model it could also be replaced by more sophisticated transformations. This would not influence the final simulation results.

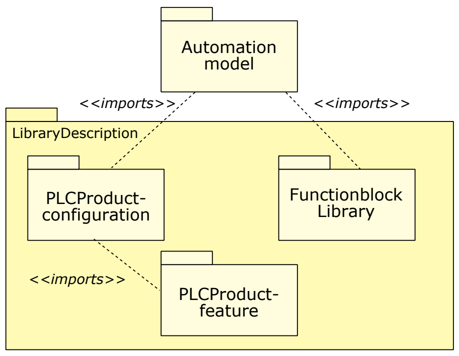


Figure 6.26: Automationmodel and LibraryDescription packages

The *Automationmodel* is furthermore split into two parts. The main part is used to specify the automation system and the influence factors. The second part is separated from the *Automationmodel* to provide reusable elements like PLC settings, configurations, and Function Blocks in Libraries. Figure 6.26 shows the relations between the different models and packages. The Automationmodel references the LibraryDescription model which contains three packages to structure the model elements. The *FunctionblockLibrary* package contains all classes to model (reusable) Function Blocks, their parameters and behavior.

To specify a PLC, the classes provided in the *PLCProductconfiguration* are used. Currently, this package only contains a single class (*PLCProductconfiguration*) to specify PLC specific properties like name, vendor, type, processing rate, the number of cores, and so on. However, each PLCProductconfiguration can reference a set of *PLCProductfeatures*. These features are later used to generate the according Palladio models. Therefore, represents the *PLCProductconfiguration* with its unique set of *PLCProductfeatures* a PLC specific load profile. Figure 6.27 shows such a configuration. The *PLCProductconfiguration* named *ICL_171_Rev_2.34* contains a set of *PLCProductfeatures*. Each of these features is accompanied by a transformation rule that creates or modifies the Palladio models as shown in section 6.3.2.

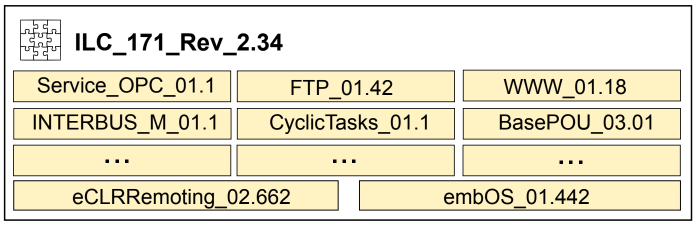


Figure 6.27: *PLCProductconfiguration* containing a unique set of features

6.3.2 Transformations

The models in Papyrus, the *Automationmodel* and the five Palladio models (repository, usage, allocation, system, and resourceenvironment) are based on the EMF technology. This allows choosing from a wide range of Model-To-Model approaches provided for Eclipse. To select a fitting transformation technology, Lehrig [Leh12] provides a decision-tree. To enable an easy, textual modification of the transformation rules and an automation system affine target audience, we choose QVT-Operational (QVTO) [Obj11b]. QVT-O is an imperative language designed for writing unidirectional transformations. It is similar to programming languages like Java and its text-based specification allows an easy customization of its transformation rules. Additionally, QVTO supports QVT-BlackBox operations for invoking external code. This allows the specification of complex calculations that are used to determine the ResourceDemand for Services, fieldbusses, or Function Blocks. Alternatives to QVT-O are ATL citeATL, TGG [Sch95], or XTend [Ecl17d] were not covered in Lehrig's work.

The following subsections introduces the two necessary transformation steps bottom up from the Automationmodel to the six Palladio input models and from the CONSENS4SysML to the Automationmodel.

Automationmodel to Palladio

One of the initial design goals of the Phoenix Contact project was to create the transformations and generation process as modular and flexible as possible. Therefore, the transformations are developed to be interchangeable and independent of each other up to a certain degree. A transformation contains mapping rules that specify, how an element from the Automationmodel is transformed into one or many elements in the Palladio models (see Figure 6.16 in section 6.2.3). A root transformation rule associated to the PLCProductConfiguration, creates the five Palladio models and the additional Taskmodel. Afterward, one or more follow-up transformations are executed that incrementally build up the whole automation system in Palladio. Figure 6.28 shows an exemplary setup of these QVTO based transformations. On the left side, an abstract Automationmodel has been depicted. It contains a PLC, CyclicTask, Program, and OPC-Server. The first transformation (*ILC_171_Rev_2.34*) creates the six models shown on the right side. Each subsequent transformation creates or modifies the existing model to add specific elements to it. In this figure, only exemplary repository elements are shown. A benefit of this approach is, despite the ability to create new PLC products more easily, that the model creation and debugging process is much more simpler than with one, complex transformation that creates all models and relation.

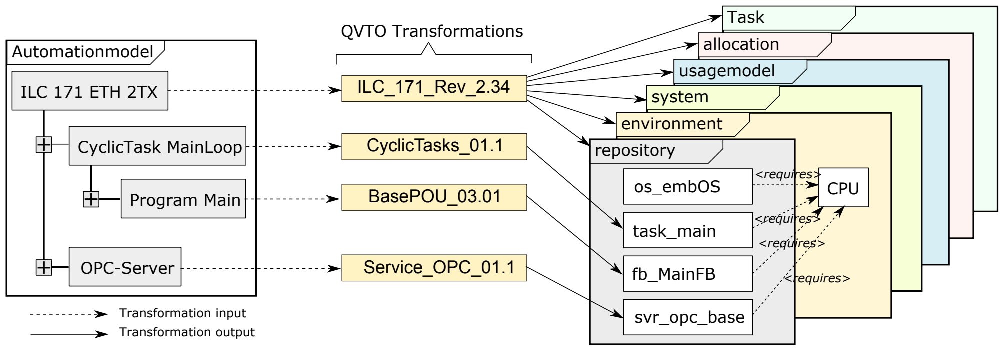


Figure 6.28: A set of transformations generating parts of the model

Another advantage of using QVTO transformations to generate the Palladio models, is the possibility to easy modify attributes and configurations. The listing 6.1 shows an excerpt from the transformation *embOS_baseload.qvto*, which contains several global variables. Changing, for example, the *priority* of related processes can be done by just setting a new integer value. The same applies to the *baseloadPercentage* and *deviation* which influence the creation of the ResourceDemand. The listing also shows two exemplary QVTO mappings used to create a resource demand in the *Repository* model.

The upper mapping is used to create a *ParametricResourceDemand* element that sets a resource demand for the CPU. Please note that the *"@repo"* at the end of the mapping signature has been removed to fit the page. This tag is used to define one of the six models this element is created in. To finalize the *ParametricResourceDemand*, a *PCMRandomVariable* must be created that specifies the actual literal of the demand. For this, the second mapping is used which builds the specification string based on the *processingrate* of the PLC, the *baseloadPercentage* and the set *deviation*.

Listing 6.1: Excerpt from the embOS_baseload.qvto transformation rule

```
// ------ Operatingsystem (embOS)
property prefix="EmbOS_Baseload";
property baseloadPercentage:Real=0.05;
property deviation:Real=0.05;
property priority:Integer=250;

mapping PLC::createRD():pcmMM::seff::seff_performance::
    ParametricResourceDemand{
  var resrepo=pcmMM::resourcetype::ResourceRepository.allInstances();
  var procresTypes=pcmMM::resourcetype::ProcessingResourceType.
      allInstances();
  var prt=rtrepo.objects()[pcmMM::resourcetype::
      ProcessingResourceType];
  requiredResource_ParametricResourceDemand := prt->any(entityName=
      "CPU");
  specification_ParametericResourceDemand:=self->map createPlcRDVar()
      ->any(true);
}

mapping PLC::createPlcRDVar():pcmMM::core::PCMRandomVariable@repo{
  var spec=(self.plcProductConfiguration.processingrate*
      baseloadPercentage);
  var rangeo1 : Real =  (spec*(1-(deviation)));
  var rangem1 : Real =  (spec*(1-(deviation/2)));
  var rangem2 : Real =  (spec*(1+(deviation/2)));
  var rangeo2 : Real =  (spec*(1+(deviation)));
  specification:="DoublePDF[("+ rangeo1.toString()+";0.00000000)("+
      rangem1.toString()+";0.05000000)("+rangem2.toString()
      +";0.90000000)("+rangeo2.toString()+";0.05000000)]";
}
```

AIM to Automationmodel

This transformation is used to gather all the information scattered in the annotated CONSENS model and generate the intermediate Automationmodel. The input for this transformation is the .uml-file created by the Papyrus editor. The Papyrus (meta-)model is also based on the Eclipse EMF technology, which allows using QVTO just like in the previous section. The transformation also contains a set of mappings that will create one or more target elements in the Automationmodel. An excerpt of these mappings is shown in listing 6.2. The upper mapping is used to create CyclicTask elements when provided an UML Property. Properties of the target CyclicTask element like *cycletime* or *priority* will be set according to the specifications given in the annotated CONSENS model. Afterward, all properties are checked if they have the applied stereotype “AIM::pou::Program”. These elements are further provided as an input to the mapping *Type2Program* which creates the according Program elements.

Listing 6.2: Exemplary mappings to generate the Automationmodel

```
mapping Class::Property2CyclicTask() : am::CyclicTask{
 name:=self.name;
 cycletime:=self.getValue(self.getAppliedStereotype("AIM::task::
      CyclicTask"),"cycleTime").oclAsType(Integer);
 priority:=self.getValue(self.getAppliedStereotype("AIM::task::
      CyclicTask"),"priority").oclAsType(Integer);
 self.ownedElement[uml::Property]->forEach(prop){
  if(prop.type.getAppliedStereotype("AIM::pou::Program")!=null) then{
      programs += prop.type.map Type2Program();
    } endif;
   }
}

mapping Type::Type2Program() : am::Program{
 name := self.name;
 executiontime := self.map Type2WorstCaseExecutionTime();
 self.ownedElement[uml::Property]->forEach(prop){
 if(prop.type.getAppliedStereotype("AIM::pou::FunctionBlock")!=null)
      then {
 functionblockUsages += prop.map Property2FBUsage();
 }endif;
};
```

6.4 Summary

In this chapter, the different models and transformation steps have been explained. Afterward, the AIM UML profile that maps the identified influence factors to the System Engineering models based on SysML4CONSENS has been introduced. Section 6.2 listed some requirements and related work before detailing the input models for Palladio, the selected analysis approach for this thesis. To generate the simulation models, a semi-automatic toolchain based on QVTO transformations has been presented. The concept and details of these transformations, as well as the intermediate Automationmodel, have been explained. It is now possible to automatically generate the input for the Palladio performance simulation from the initial Systems Engineering models.

CHAPTER 7

Evaluation

This chapter provides an evaluation of the concepts developed in Chapter 5 and their realization in Chapter 6. The goal is to validate whether the provided influence factors and the proposed modeling approach can be used to conduct a performance prediction based on Systems Engineering models. Figure 7.1 shows the steps and artifacts performed in this evaluation. The UML profile is used to create an annotated Systems Engineering model. In combination with the developed Simulation models and transformations, this model is used to run a performance simulation (step 6). To evaluate the results of this simulation (step 7), a performance prototype (see Chapter 2) is created and measured. The comparison of the predicted and the measured utilization of a PLC shows that the approach is sufficiently precise to validate the Systems Engineering models in the early development stages. The maximum deviation is around 8,7%, which is still below the desired prediction accuracy of 20 percent.

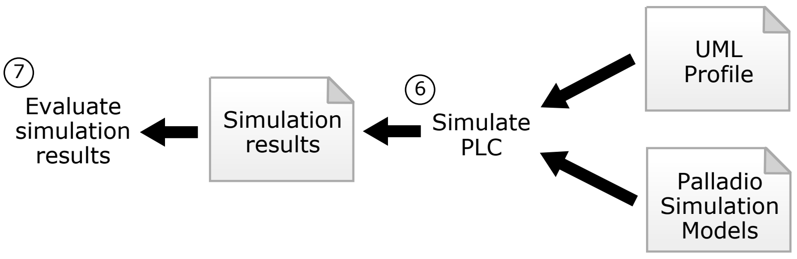


Figure 7.1: Steps and artifacts of this chapter

7.1 Evaluation Process

According to Freiling et al. [EFR08] there are several types of validations that can be used to evaluate an approach. A Type 0 validation requires a tool, approach or metrics that can be compared to each other showing that the approach is working in general. This validation has already been provided by identifying the influence factors in Chapter 4 and the following realization using the UML profile and Palladio as a simulation framework. The Type I validation requires a comparison of predictions and measurements, that show that the values, to some degree, conform to the observed reality. The Type II validation focuses on the ease of use or applicability of the approach. It is therefore checked, whether the developers can (easily) apply the proposed development process and modeling tools to specify all influence factors. This also includes the analysis and meaningful interpretation of the prediction results.

The goal of this evaluation is to verify, that the developed models, tools, and transformations can be used to predict the utilization of an automation system in the early development stages. For this, a Type I validation has been be conducted. The Goal Question Metric method [Bas92] is used to state the following GQM goal definition template.

Goal:
Analyze the modeling and simulation of influence factors **for the purpose of** validating the simulation **with respect to** the prediction accuracy **from the viewpoint of** an automation system developer **in the context of** a simplified turbocharger example.

Following GQM, a question and corresponding metrics are derived. This evaluation is focuses on the overall utilization of the PLC, which is expressed in a percentage value. This percentage value is set as the metric. The question is, whether the prediction accuracy deviates more than 20 percent between prediction and measurement.

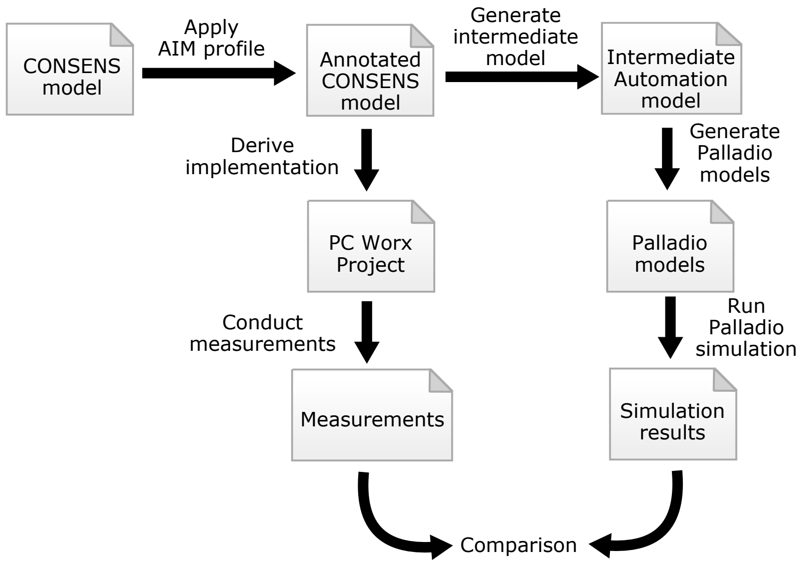


Figure 7.2: Evaluation steps to compare simulation with measurement

Figure 7.2 shows the steps that are performed to determine the prediction accuracy with measurements taken directly from the PLC. First, existing *CONSENS models* are extended by the *AIM profile* and the influence factors are added to the System Engineering model. This *annotated CONSENS model* is the basis for the following implementation as well as the generation of the *Intermediate Automationmodel.* This model provides additional information and is transformed to Palladio models for simulation purposes. This simulation is performed by the SimuCom framework. The implementation of the IEC code and the configuration of the PLC is done in *PC Worx*, the Engineering tool for Phoenix Contact PLCs. The chosen PLC (ILC 171 ETH 2TX) allows the measurement of the overall utilization which is compared to the simulation results in the last step.

The following sections detail the context of this evaluation and the setup of the performance prototype. They provide necessary information on the used Systems Engineering models, the set of influence factors that are evaluated, and how the metrics are measured on the automation system. Afterwards, the results of the measurements and predictions are presented, followed by a short discussion of the gathered results. Before giving a final summary, the threats to validity are discussed.

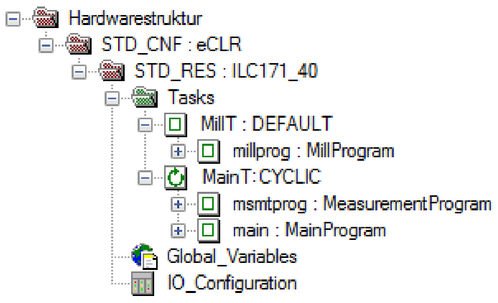
7.2 Evaluation Context

For the evaluation, the Turbocharger production system introduced in Chapter 3 and a simple test case for the IdleTask are used. These two scenarios are based on the same setup (see Section 7.2.1) and are described in detail below.

Simplified Turbocharger Test
The CONSENS models provided in that Chapter 3 are extended by the AIM profile developed in Chapter 6, adding influence factors, parameters, and settings. This information can be used to automatically generate Palladio models for a performance simulation.

However, the example CONSENS models include all influence factors and a broad range of Function Blocks, Functions, Services, and Operations. Due to the limited performance of the ILC ETH 2TX PLC, the complete running example from Chapter 3 cannot be used, but only a subset of it. The available PLC is not capable to run multiple Function Blocks with high loads and has the full range of services modeled. For example does the ILC 171 only provide an OPC-Server instead of an OPC-UA-Server. To create performance models for the simulation, a basic profiling of each influence factor is necessary. However, the only available PLC with the necessary in-depth profiling mechanism was the ILC 171 provided by Phoenix Contact. Therefore, the overall number of Function Blocks has been reduced to eleven in contrast to the 36 used in the running example (see Appendix A) to be able to run on the ILC 171. The Function Blocks are executed in three programs, triggered by two tasks. Also, the older OPC server (instead of the OPC-UA server) doesn't provide a remote operation access or file transfer. In consequence, the two services accesses are removed as well from the evaluation model. Figure 7.3 shows an excerpt of the project structure of PC Worx, the Engineering tool for Phoenix Contact PLCs. The following list details the setup for the performance prototype and simulation models.

- **Eleven Function Blocks** are defined for this example, each with varying ExecutionTime specifications. The Function Blocks are instantiated in three Programs and executed in their associated tasks.



```
Hardwarestruktur
  STD_CNF : eCLR
    STD_RES : ILC171_40
      Tasks
        MillT : DEFAULT
          millprog : MillProgram
        MainT: CYCLIC
          msmtprog : MeasurementProgram
          main : MainProgram
      Global_Variables
      IO_Configuration
```

Figure 7.3: Screenshot of the project setup in PC Worx

- **Three Programs** are used that instantiate the Function Blocks. Each Program is set up with an execution time, also specified as a fixed WCET. The Programs are named *MillProgram*, *MeasurementProgram* and *MainProgram*.
- To execute the Programs and Function Blocks, **two Tasks** are set up. First, the *MillTask* is an IdleTask to incorporate the custom Scheduler for the IdleTask. The second task is a cyclic task called *MainTask* with a cycle time of 150 ms. The MainTask has a higher priority than the MillTask.
- The annotated CONSENS model contains an **FTPServer** which is periodically accessed by an FTP-Client to download logfiles. This client will connect to the server every 5000 ms and request a 100 kb file for download.
- An **OPC-Server** is used to provide 1000 integer variables that will be read by an OPC-Client. The client has a set refresh interval of 1000ms.
- The **Interbus IO** will be set up with fixed cycle time. Just the *MainProgram* will write 8 digital outputs. Setting a fixed cycle time is a common action to avoid letting the PLC automatically determine the best cycle time which could lead to an oversampling of the fieldbus. For this example, the Interbus cycle time is still 15 times faster than the cyclic *MainTask*. Increasing the cycle time would lead to a hard measurable load induced by the fieldbus.

Idle Task Test

The second scenario focuses on the runtime behavior of the Idle Task (see Section 4.3.4). This common task type in automation systems executes the associated Program instances in an endless loop. As described in Section 6.2.3 is a special scheduler used that limits the CPU utilization of the PLC to 60%. Therefore, the second evaluation context is build up without external influences and service accesses and just contains a simple Program in an Idle Task. It is evaluated, whether the utilization will settle at the specified limit of 60%.

- One **Program** is used that contains a single Function Block. The execution time of the Program is increased from 1ms to 500 ms to create utilization on the PLC. The value is changed after each set of measurements.

- To execute the single Program, a **IdleTask** is set up and has has no further settings. The in-depth parameters of the IdleTask for the ILC 171 ETH 2TX PLC are given in Section 6.2.3.

7.2.1 Setup of the Performance Prototype

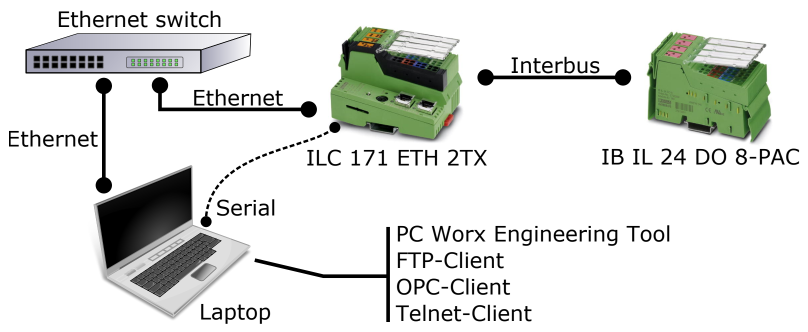


Figure 7.4: Setup of the performance prototype

In this section, an overview over the implementation details of the performance prototype and the conducted measurements is given. The model and its influence factors are simplified to create a runnable setup for the used PLC.

Figure 7.4 sketches the setup for the performance tests. The PLC used for the execution of IEC code is a *ILC 171 ETH 2TX* small scale PLC from Phoenix Contact. The PLC provides an Interbus IO system which is used to connect a bus coupler *IB IL 24 DO 8-PAC*. This bus coupler can support up to 8 digital outputs. The PLC provides also an OPC-Server, FTPServer and Webserver for an HMI client.

A Laptop is used to program and configure the PLC with PC Worx. It is connected to the PLC via a standard Ethernet switch. The Laptop also runs the FTP-Client which used to download log files from the PLC as well as an OPC-Client that accesses variables. The special debugging firmware of the ILC 171 provides the possibility to access internal states and status information via a serial link. The PC can read the data over this link via the telnet protocol. The data is used to gather detailed information about processes, threads and the overall utilization of the PLC.

The performance prototype does not provide Programs and Function Blocks with code based on the actual Turbocharger example. The original code has been restricted by ELHA. Therefore, new code to represent the functions of the original software needed to be implemented. To emulate the workload of the original code, *LoadGenerator* Function Blocks are used that will stress the CPU and create a specific utilization of the PLC. Listing 7.1 shows the Structured Text of this LoadGenerator. For a given time interval, a sine-function is executed - the calculated values are not used and discarded. The interval is calculated based on the *PLC_SYS_TICK_CNT* signal, which is the most precise timer the

ILC 171 provides to IEC software developers. This poses a problem concerning the minimum load that can be simulated by a LoadGenerator. The timer has only a precision of 1ms, which means that all specified ExecutionTimes must be a multiple of 1ms. For this reason, the example contains no ExecutionTimes with less than 1ms. Real world examples usually contain Function Blocks with far shorter execution times.

Listing 7.1: Structured Text code of the LoadGenerator Function Block

```
IF Busy THEN
  LastVal := PLC_SYS_TICK_CNT;
  REPEAT
    Real_Val := Sin(Real_Val * 2.3);
    LoadProcessigTime := PLC_SYS_TICK_CNT - LastVal;
  UNTIL LoadProcessigTime >= ProcessigTime
  END_REPEAT;
END_IF;
```

To emulate the usage of the fieldbus, a simple counter has been implemented that provides a Byte value which is split up onto the eight digital outputs.

Figure 7.5 shows the Function Block Diagram that implements the Program *MainProgram*. Depicted are the different Function Block instances for the Turbucharger example (green) including the LoadGenerator instance. The Function Blocks colored in red are used to generate the output for the Interbus fieldbus. The value of the counter is converted to a Byte variable named *Zaehlwert* and mapped onto the IOs.

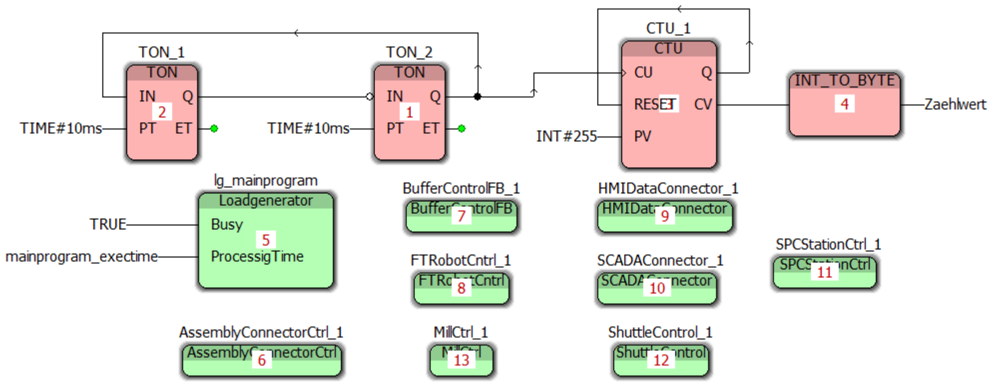


Figure 7.5: Function Block Diagram of the MainProgram

To measure the PLCs utilization, a terminal connection is used in conjunction with a specific debugging firmware provided by Phoenix Contact. In contrast to a regular firmware, this allows the debugging version to send commands via the telnet protocol that are interpreted on the PLC and console output send back the client on the laptop. One of these commands enables the profiling of the PLC. For a set period, the PLC logs every task, process, memory consumption, execution time, and overall utilization of the CPU. The log file shown in the listing has been obfuscated to hide vendor specific information.

Listing 7.2: Serial console output (obfuscated/modified)

```
Wait for end of Profiling (5sec)
[181] embOS task list
[182] ----------------
[nr] task       state T pri stack:size/free/used      name
[ 0] 0x0250b8d0 Sema  250 s=0 f=0 u=-1% CPU-Load=0.00%   aThread
[ 1] 0x029a6640 Delay 240 s=0 f=0 u=-1% CPU-Load=11.00% Systick
...
[85] 0x0250bfc8 Delay   2 s=0 f=0 u=-1% CPU-Load=6.20%  aProcess
[183] ----------------
[184] Used Memory for Stacks: 0 kB
[185] Total CPU Load: 80.00%
```

For the evaluation of multiple influence factors including the access of services like the OPC-Server and the FTPServer, a tool chain has been developed. This tool chain has three functions. First, it allows the generation of loads with tools like *wget* to automatically download files. Second, it uses the terminal connection to execute a profiling on the PLC. Third, it collects all log files, parses the data, and calculates mean, median, minimum, and maximum values - if applicable. These processed measurements are used for the evaluation.

7.3 Evaluation Results

After the context for the evaluation has been set up and some details of the performance prototype have been introduced, the measurements are compared against the predictions. This section lists the measurements for the simplified Turbocharger example and the dedicated IdleTask scheduler scenario. The first scenario covers a complete example with various factors. The second scenario is used as a dedicated test to validate the custom scheduler created for the IdleTask.

Simplified Turbocharger Scenario
The annotated CONSENS models have been transformed into Automationmodels. They provide further means for configuration, like the selection of a PLC and its load profiles. In the following step, the Automationmodel is transformed into Palladio models and a simulation with Simucom is executed. Simucom and the sensor framework (as part of the Palladio Framework) provide means to analyze the simulation results. For the Turbocharger example, a time span of 20 seconds has been simulated. This took round about one and a half minute.

Table 7.1 provides an overview of the measurements and predictions. Overall, five independent simulation runs have been carried out. The predicted average utilization of the PLC can slightly vary, due to influence factors like the Operating system which are not modeled with a fixed WCET but a RandomIntervalExecution. Therefore, multiple simulations have been run based on the same input model. To obtain a valid measurement of the real PLCs utilization, multiple measurements are performed and processed. Between each measurement, a pause with a length of five seconds is made to collect the data from the PLC without interfering the following measurement. The table shows simulation results in the first row, followed by the according measurements below. The last row lists the absolute prediction error.

Table 7.1: Comparison of predicted and measured CPU utilization

	#1	#2	#3	#4	#5
Prediction	80,6%	80,4%	80,3%	80,4%	80,4%
Measurement	80,0%	80,2%	79,9%	81,5%	83,1%
Difference (abs)	0,6%	0,2%	0,4%	1,1%	2,7%

Idle Task Scenario
To further check whether the concepts and implementation of the IdleTask custom scheduler are correct or not, this second scenario is realized. It consists of only a single IdleTask containing a simple Program. The execution time of this program can be varied by setting a global variable used by the LoadGenerator to perform sine-functions accordingly. For the test, the execution time is increased in steps from 1ms up to 500ms. Higher execution times lead to a watchdog due to an overload of the PLC. Table 7.3 lists the average CPU utilization in percent of the simulation (Pred.) and measurements (Meas.). The absolute difference (in percentage points) between the values is given in the last row.

Table 7.2: CPU utilization with increased Program execution times

	1 ms	2 ms	5 ms	10 ms	20 ms	30 ms	50 ms	100 ms	500 ms
Pred.	44,1%	52,2%	75,9%	74,4%	74,4%	74,6%	73,9%	74,1%	77,0%
Meas.	52,8%	56,0%	73,9%	71,9%	71,2%	71,0%	71,3%	73,7%	72,1%
Diff.	8,7%	3,8%	2,0%	2,5%	3,2%	3,6%	2,6%	0,4%	4,9%

Figure 7.6 visualizes the predicted utilization (Sim) and the according measurements on the performance prototype (PLC). The graph, as well as the table, show that the differences are marginal, despite the executions with 1ms and 2 ms generated load. The IdleTask scenario shows, that the embedded scheduler of the Palladio simulation works as expected. The overall utilization settles at 75%. This is the load induced by the Program (limited to 60%) on top of the background utilization of the Operating System and runtime environment. The much lower utilization at execution times below 5 ms is a result of the scheduler algorithms that dynamically calculate the wait time between task executions. If the calculated wait time is lower than a set limit, a minimal wait time of 4ms is forced. This leads to the observed lower utilization at 1 ms, 2 ms, and 4ms execution time settings.

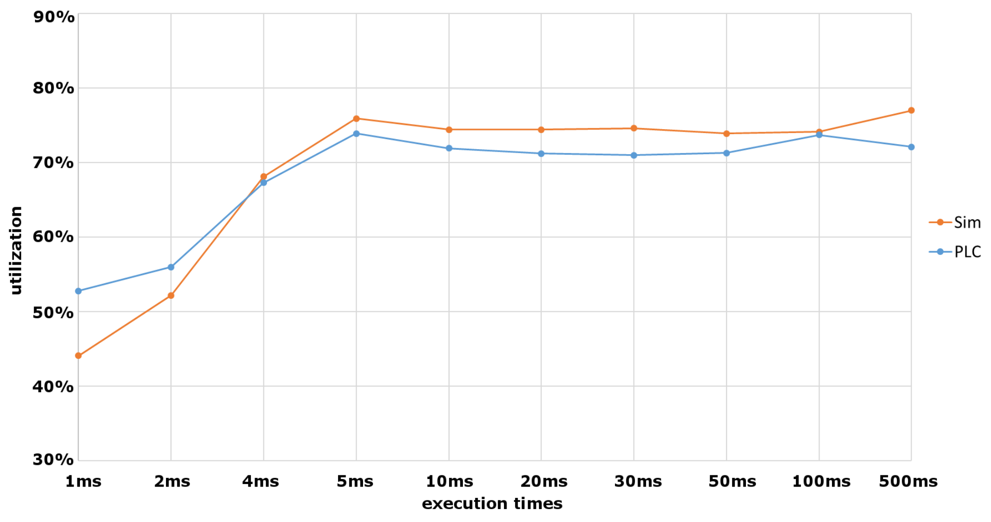


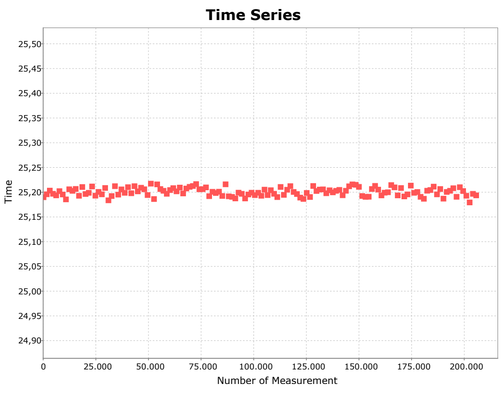
Figure 7.6: Visualization of the IdleTask run

7.4 Discussion of the Results

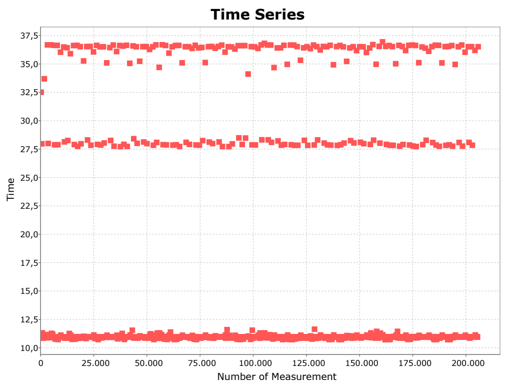
In this section, the results of the simplified Turbocharger example and the IdleTask will be discussed. Both scenarios reveal that the predicted utilization of the PLC is very close to the measured one. The maximum deviation is around 8,7%, which is still below the desired prediction accuracy of 10 percentage points.

Simplified Turbocharger Scenario
The Table 7.1 listing the results of the simplified Turbocharger example, shows that the predicted values are very close to the measurements. There are several causes for this effect. First, the simplified CONSENS model contains just a few influence factors. Furthermore, are ExecutionTimes only modeled as fixed WCET and not with random values. A reason for this decision is the implementation of the performance prototype, which does not provide adequate random generators to vary inputs for the load generators. Additionally are all services accessed in fixed intervals instead of varying/random access patterns. Finally, are all resource usages for the (operating system) background tasks measured and executed per SystemTick. All these points lead to an evenly distributed load, reducing spikes and further simplifying the simulation.

Palladio provides even more details about the system under investigation. Figure 7.7 shows two times series diagrams generated by Palladio. Diagram 7.7a shows the number of measurements on the x-axis and the time for the execution of the MainTask including all Programs and Function Blocks. The execution times are all in a tight corridor around the 25 ms execution time. This flickering can be explained by the high priority operating system tasks like the *SysTick* and the Interbus Master process, which pause and later continue the MainTask. The right Figure 7.7b shows the time series for the MillTask. This task has a lower priority than the cyclic *MainTask* and other internal tasks like the *SysTick*, *Interbus*, *eCLR-Process*, *Remoting-Services*, and more. The preemption will therefore not only extend the duration of the execution but will also increase the following wait time of the idle task. Therefore the execution time is flickering much more heavily than the one of the cyclic task.



(a) MainTask (Cyclic)



(b) MillTask (IdleTask)

Figure 7.7: Times series graph for the two tasks

Idle Task Scenario

The simulation results of this scenario are shown in the Table 7.3 and visualized in Figure 7.6. The graph as well as the table show that the differences are marginal, despite the executions with 1ms and 2 ms execution times. A reason for these deviations could be operating system tasks and copy actions that could not be investigated in detail for such small execution times. Another reason for the deviation of predicted and measured utilization for the short execution times (1 ms and 2 ms) could be the overhead of the profiling mechanism. Furthermore, shows the graph an upper limit of the utilization at round about 75%. This effect reflects the current implementation of the scheduler, which holds the execution of the IdleTask to provide a buffer for load spikes that would otherwise lead to a watchdog. The profiler captures extensive information about memory consumption and process execution of each thread. This will also put a load onto the CPU which is likely higher in scenarios where the task context is switching more often.

Conclusion

The results of the Simplified Turbocharger and IdleTask scenarios showed the general applicability of the approach. The comparison of predicted and the measured utilization of a PLC shows, that the approach is sufficiently precise to validate the Systems Engineering models in the early development stages. The maximum deviation is around 8,7%, which is still below the desired prediction accuracy of 20 percent. The overall process of annotating existing SysML4CONSENS models and their transformation into Palladio simulation models has been evaluated. A subsequent simulation supports the developers of an automation system to understand and estimate the future utilization of a selected PLC. This will result in less changes in the integration phase and decreases the time to market and overall development costs.

7.5 Threats to Validity

After presenting the results of the comparison between measurements and predicted values in the previous section, some threats to the validity are now further detailed.

Size and detail of the model. The example introduced in Chapter 3 is taken from a real world industrial automation system. However, the CONSENS models needed to be simplified and obfuscated. They still cover all identified influence factors and have a reasonable size to support the proposed process and modeling approach. But the models and factors used for the evaluation had to be further stripped down. A reason for this, are the limited capabilities of the PLC used for the performance prototype. The PLC has a much lower CPU frequency and less memory than the PLC used in the original ELHA production system. Also, the actual number of IOs could not be recreated in the performance prototype. This might pose a threat to the validity since the performance prototype only presents a fraction of a real world example. However, the comparison of measured and simulated utilization is both based on the performance prototype and its according scenario, showing that the early validation is feasible for small examples. In the future, more in-depth models and other PLCs should be evaluated to further prove the general applicability and scalability of this approach.

Falsification through profiling. The measurements taken from the PLC use a specific profiling mechanism implemented in a special debug firmware. Therefore, side effects and an additional load on the CPU are induced, due to the profiling itself. This blurred not only the comparison in the Evaluation chapter but also leads to a difference in the (Palladio) load profiles for the given PLC. This could be one explanation for the differences between the predicted and measured values in Figure 7.6 at small execution times.

Detail of measurements. As already mentioned had several of the operating system and other firmware specific tasks to be combined into a single or set of load profiles. The reason for this was in most cases the minor utilization induced by these tasks or the frequency they are executed. In addition to this, provided the profiling mechanisms just fixed sessions of five seconds, in which the task or thread specific loads were listed with their mean value. Therefore, it is not possible to model each load on the PLC in detail. This further lead to an evenly distributed load of multiple tasks, making detailed predictions less accurate.

Limitations of the performance prototype. The ExecutionTimes modeled only as fixed WCET and not with random values like RandomSet or Bounded. A reason for this decision is the implementation of the performance prototype, which does not provide adequate random generators to vary inputs for the load generators. Additionally, has the Interbus Fielbus been set to a fixed cycle time and does only include one bus coupler with eight digital outputs. This setup is therefore not representative of real world applications.

Knowledge of influence factors in the early development stages. The Turbocharger model has been extensively investigated and influence factors modeled based on knowledge of the actual implementation of the firmware. This leads to detailed specification of influence factors like Service access intervals or WCET for Function Blocks. This knowledge is not available for the development of a ground up new automation system. The evaluation shows, therefore, the general applicability of the approach, but not under real world conditions. However, the models and influence factors are tailored towards the rough specification of execution times and loads, or the goal to set upper limits for the software developers and engineers to adhere. These two goals are achieved by the current level of realization and therefore support the hypothesis of this evaluation. Still, this threat to validity should be further investigated by conducting in-depth user studies as a Type II evaluation.

Availability of load profiles. This small, exemplary evaluation showed the applicability of a realization with Palladio simulation models. The load profiles for the ILC 171 ETH 2TX PLC could be created due to the in-depth knowledge of the vendor. This level of detail might not be (easily) achieved for other PLCs or different vendors. To create these load profiles, the PLCs must either support extensive profiling mechanisms or the vendors must provide sufficient information to create sufficiently precise profiles. For this thesis, no other available PLC provided the necessary data nor possibilities to create a secondary load profile. Therefore, it is unsure if the developed concepts can be transferred. However, the modeling and specification of influence factors is generally applicable to other PLC. They might provide different services and unique load profiles, but the identified influence factors are non-vendor specific.

7.6 Summary

The evaluation of the proposed development process, the modeling approach, and simulation with Palladio showed the general applicability of this thesis contribution. However, there are several threats to validity that must be considered. For a majority of these threats, a further investigation and/or user study will clarify the general applicability of the proposed approach. Examples are the simplified model of the automation system with a small number of influence factors, or the performance prototype with limited capabilities. Despite this, the results of the comparison between simulated and measured PLC utilization showed a promising initial approach to predict the performance of an automation system. This **confirms the hypothesis**, that the developed approach can be used to **predict the performance in the early development stages of an automation system**.

CHAPTER 8

Conclusion and Summary

Designing and scaling complex, networked automation systems is a challenging task. Already in the early stages of the development, developers have to consider different kinds of influence factors that have an impact on the overall system performance. Estimating the throughput the average workload of a PLC, deploying software onto resources, or assigning sensors and actuators to PLCs are difficult tasks when coping with complex systems. If performance bottlenecks are detected too late, costly corrections may follow and the commissioning can be significantly delayed.

For the domain of industrial automation there already exists several approaches to predict the behavior and quality of service attributes of a system under development. However, these approaches usually differ in their level of detail and their notation towards the domain of industrial automation systems. Additionally, these approaches lack the possibility to capture automation-specific influence factors on a high level of abstraction which is necessary for the use in the early phases of the development. A common set of factors which cover the core automation specific influences on different aspects of the system is not available.

The objective of this thesis was to tackle the two main problems that have been identified. First, the lack of a generally applicable set of influence factors which can be used for the validation of automation systems in the early development stages. This problem has been solved with the contribution of an extensive list of influence factors that impact an automation system as shown in Figure 7.1. It is the result of an identification step (1) and based on different artifacts and related work. The second problem was the absence of a method to capture these influence factors in Systems Engineering models and an appropriate process to guide automation system developers through their specification. For this, the Automation Influence Model (2) and a process (3) haven been developed. To evaluate these contributions (6 & 7), a UML profile (4) and simulation models (5) have been created.

In the following, the main contributions of this thesis are summarized in Section 8.1. Afterward, the core benefits of this approach are highlighted in Section 8.2. Finally, the limitations and remaining questions of this approach are discussed and opportunities for future work are presented.

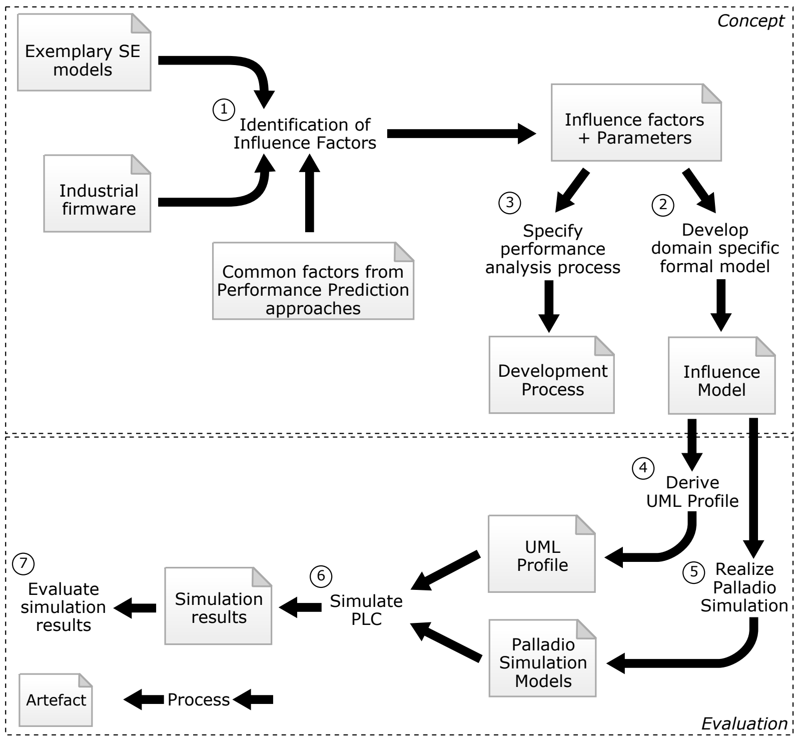

Concept
Exemplary SE models
Industrial firmware
Common factors from Performance Prediction approaches
1 Identification of Influence Factors
Influence factors + Parameters
3 Specify performance analysis process
2 Develop domain specific formal model
Development Process
Influence Model
4 Derive UML Profile
UML Profile
5 Realize Palladio Simulation
Palladio Simulation Models
6 Simulate PLC
Simulation results
7 Evaluate simulation results
Artefact
Process
Evaluation

Figure 8.1: Performed steps and created artifacts to perform automation specific performance predictions

8.1 Results and Conclusions

In this section, the results of this thesis are briefly described. They are separated into the two main contributions “C1: Identification of Influence Factors” and “C2: Method for Modeling Automation System Influence Factors”.

C1: Identification of Influence Factors
A broad range of approaches to model and predict quality of service attributes of automated systems focus on a specific set of influence factors. A complete list of such influence factors that impact the overall automation system is not available. However, this is a prerequisite for an analysis of the system during the early development stages, where specific details are not yet available.

The first contribution of this thesis in Chapter 4, was the gathering of influence factors for an automation system that will impact one or more quality of service attributes. For each factor is discussed whether it is available in the early development stages, what assumptions have to be made by the developer, its overall impact on the system, and which parameters need to be taken into account. The gathered list is the basis to extend System Engineering approaches like CONSENS by automation specific information. Models extended with this information can be used to analyze, respectively predict, the performance of the system before costly and time-consuming changes must be made. To find the necessary factors, three primary sources have been taken into account:

Exemplary Systems Engineering models of automation systems, the firmware of PLC vendor Phoenix Contact, and common factors from existing performance prediction approaches.

As a result, the key influence factors and parameters that impact the performance of PLC used in an automation system have been identified. This is a necessary prerequisite to create the formal Automation Influence Model and the according development process.

C2: Method for Modeling Automation System Influence Factors
Current Systems Engineering methods support the development of automation systems, but domain-specific performance-relevant information is neglected. Therefore, the second contribution of this thesis was to provide a method for capturing automation specific influence factors on a high level of abstraction that is applicable for the use in the early phases of the development.

For this, a method has been developed that allows the automation system developers to specify influence factors integrated into System Engineering models (see 5). The two parts of this method are a formal model to capture the various factors (5.1) and a process to guide the developers through the specification (5.2). The formal model incorporates domain-specific elements and therefore allows developers to use their well-known terminology. In this thesis, CONSENS has been selected as an exemplary Systems Engineering approach on which this formal model will be applied. To further evaluate the process and model, an UML profile that extends the SysML4CONSENS-Profile has been created. This profile contains stereotypes to provide additional information which can be used to annotate CONSENS elements by automation specific influence factors. The process to guide developers through the specification of factors and parameters will define, at which point in the development process of an automation system a performance analysis should be carried out and what information needs to be gathered before starting the analysis. As a basis for this development process, an existing CONSENS process for the specification of software requirements has been selected.

As a result, the developers of an automation system are now able to use the process to specify important influence factors that impact the performance of a selected PLC. This will allow them to evaluate the system design in the early development stages of the system and therefore reduce costly and time consuming changes in the later phases.

Realization
The developed process, the formal Automation Influence Model, and the UML profile have been evaluated. For this, an exemplary Turbocharger production system has been modeled and used for a subsequent analysis. To do so, the Palladio Component Model and its simulation engine SimuCom have been selected. Palladio provides a good interface to access and generate input models,

is easy to use, allows a visual inspection of the simulation models, and supports a fast and reliable performance simulation. The capabilities of Palladio had to be extended to support the Idle Task as an automation domain specific requirement. Different load profiles have been created to finally run a simulation and predict the utilization of a PLC in the context of the Turbocharger example. The predicted utilization of the PLC has been compared to measurements taken from a performance prototype which resembles the basic characteristics of the exemplary automation system. The result of the simulation showed a general applicability of the two contributions of this thesis.

8.2 Benefits

The results of this thesis provide a wide range of benefits to developers of automation systems during the early development stages. This section highlights selected results and how they improve the design of complex automation systems. This includes the developed models and process as well as the tool and additions for the realization and evaluation.

- The integration of automation specific influence factors in System Engineering models allows developers to **discuss impacts** and **synchronize changes** that span multiple disciplines. This will help to **find errors**, imprecise requirements, **and inconsistencies** in the early development stages, further **reducing costly** and time-consuming **changes**. By specifying execution times and other properties on various parts of the automation system, the **discipline specific developers** will have **requirements** and **limits to adhere**. With CONSENS, a well-known Systems Engineering model has been selected that has already been applied in several projects, like the introduced Turbocharger production system. The abstract and automation system specific notation should simplify the use of the UML profile, further increasing the acceptance and the results of this approach.
- The use of an UML profile to extend CONSENS **conserves the structure and information** of the models. The additional data in form of influence factors and their parameters is appended to the existing elements. This also applies to the case in which topology independent models are exchanged for the more detailed topology specific models. The TSM annotations are **added** to existing elements and **do not interfere** with the basis model. Also, by using a profile, state-of-the-art UML tools that can incorporate the automation specific stereotypes and **provide** a more **user friendly modeling environment**.
- The information provided in the annotated Systems Engineering models can be used to run a performance prediction of a selected PLC. This will enable the developers to **investigate**, whether the **performance of the PLC is sufficient** for the modeled task at hand. Changes to the used fieldbus, task structure, Programs, and Function Block will lead to different **performance properties** that can now be roughly **estimated**.

Also, decisions whether to split functionality onto different PLCs can be analyzed **before the system integration phase**. The information specified is also **usable for further analysis** with more detailed approaches. For example, is it possible to extract the network specific information from the model and use it as an initial basis for network simulator input models. The same applies for more detailed software development steps for which MARTE can be used to detail functions of the software.

- The developed process provides an initial **guide** for developers **at which point** in the development process of an automation system a performance analysis should be carried out and **what information** needs to be gathered before starting the **analysis**. For the development of complex or large Systems Engineering models, in which several system analysts are involved, this is a good basis they can build and orientate on.
- By using the Palladio Component Model and EMF-based CONSENS models, a simple but effective tool and model chain can be used. Its allows to transform the Systems Engineering models into simulation models with **minimal user interaction** and therefore further **improving** the **usability** of the approach. This also **reduces the chance** of modeling **errors**. As mentioned above, is it also possible to generate input models for different analysis approaches by just replacing the transformations at the correct point in the tool chain.
- The use of a modular system of transformations allows the **composition** of load **profiles** for different PLCs. As a result, profiles for new PLCs can **be created faster** by combining (existing) transformations. Separating the different influence factors into independent transformations increase the re-usability. This will further improve the general applicability of the approach.

8.3 Future Work

The results of this thesis raise several possibilities for future work. In the following, a few of these are listed and briefly described. In section 7.5 some threats to validity have been discussed that could be tackled in the future as well.

Extended User Studies
According to Freiling et al. [EFR08] are there several types of validations that can be used to evaluate an approach. The Type II validation focuses on the ease of use or applicability of the approach. It is therefore checked, whether the developers can (easily) apply the proposed development process and modeling tools to specify all influence factors. This also includes the analysis and meaningful interpretation of the prediction results. This has been omitted in this thesis but should be conducted in the future. Such an evaluation would help to confirm that the integration of automation specific elements in Systems Engineering models supports users to create complex automation systems. Going one step further, a Type III evaluation could be conducted. It investigates

whether a newly introduced method improves the whole development process. A similar study has been performed in [VHSFL14], where different groups of students developed a system with this new approach and without and their results and feedback have been compared.

Evaluate Larger Models
As mentioned in the threats to validity, has the used model for the evaluation been simplified and obfuscated. It still covers all identified influence factors and has a reasonable size to support the proposed process and modeling. However, using real world automation systems to evaluate the process and model should be an important next step, due to the fact that the size of the system under investigation has an impact on the evaluation [KPP+02, EBGR01]. Therefore, it must be verified, that the precision of predicted results and general applicability holds for larger examples.

Improve Tool Support
An obvious point that should not be underestimated is the overall tool support that can be improved. Currently, the existing CONSENS model has to be manually extended by the UML profile to be able to specify the different influence factors. The stereotypes in this profile also need to be manually added to the corresponding CONSENS System Elements. This is a hideous and in some cases error-prone task that can be improved by offering appropriate tool support. Also, several OCL expressions are used to inhibit the annotation of wrong elements or to enforce model consistency. However, there are still several constraints that must be implemented to create valid annotations. Visual feedback for developers indicating modeling errors in the model will further improve the overall usability of the approach.

Integrate and Analyze wider Range of Requirements
Currently, the approach focused on the analysis of the performance of a selected PLC. However, the annotation of automation specific elements allows the detection of other properties and problems as well. For example, could the maximum number of devices for a certain fieldbus be verified or service availability for PLCs types checked. A more detailed approach for specifying functional and non-functional requirements has been provided by [VHSFL14]. Also, in [JLS11] several kinds of requirements can be specified via a profile based on SysML and later analyzed by using Modelica. Being able to automatically check additional requirements would further improve the development process and help developers to find errors and inconsistencies in the early development stages.

Support for Additional Analysis Approaches
As listed in section 6.2.2 is there an increasing number of approaches to analyze various aspects of an automation system and its quality of service attributes. In this thesis, the Palladio Component Model and the SimuCom simulation have been used to predict the utilization of a selected PLC. However, it is a logical step to further include other tools and approaches to analyze different aspects of the system. Examples are the network throughput which can be simulated with OMNeT++ [Ope17] or automation control loops with Modelica [LWF08]

and TrueTime [CHL+03]. The integration of these approaches could be realized by exchanging the appropriate tools and transformations in the toolchain (see section 6.3).

Integrating and Interpreting Analysis Results
An important part of the analysis is the presentation of results to the developer. Currently the available feedback of a performance simulation is a table view that provides values for the overall utilization of the PLC. However, this could be provided back into the original model and annotated or even highlighted there. Such a visual feedback could help identifying bottlenecks. When incorporating a wider range of requirements and their validation these results could be added to the original model as well. For example, could a selected fieldbus be set to a red color in case the timing constraints (cycle time) could not be adhered or if too many devices have been added. An interpretation of the analysis results could support user to find an issue with the current design. A simple example would be a Function Blocks that uses too much CPU time and an automatic interpretation would suggest setting that Function Block in a different task.

Automatic Load Profile Creation
The creation of the individual load profiles for each PLC and influence factor is a time-consuming task. Depending on the PLC, a detailed analysis is often not even possible due to the fact that the overall utilization or process details are not available. However, some PLCs provide means to access this data and perform a detailed profiling. With automated approaches similar to [Abd00, JHHF09], load profiles could be created automatically. How this can be achieved and whether these automatically determined values are correct, must be investigated in future work.

Automatic Derivation of Domain Specific Models or Code
The approach presented in this thesis is based on analyzable and automatically processable models. This means that all inputs provided by the CONSENS models and annotated by the automation system specific influence factors can be used in the following discipline-specific development steps. For example, is it possible to collect all annotated Function Blocks and create Skeletons in a desired engineering tool. This approach has also been tackled by Vogel-Heuser et al. [VHSFL14] to generate Structured Text code for an initial software model.

Bibliography

[AADG12] Saoussen Anssi, Karsten Albers, Matthias Dörfel, and Sébastien Gérard. chronval/chronsim: A tool suite for timing verification of auto-motive applications. *Proc. Embedded Real-Time Software and Systems, ERTS*, 2012.

[ABC$^+$03] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, et al. Mathematical markup language (mathml) version 2.0 . w3c recommendation. *World Wide Web Consortium*, 2003, 2003.

[Abd00] T. F. Abdelzaher. An automated profiling subsystem for qos-aware services. In *Proceedings Sixth IEEE Real-Time Technology and Applications Symposium. RTAS 2000*, pages 208–217, 2000.

[ACS09] Saurabh Amin, Alvaro A. Cárdenas, and S. Shankar Sastry. *Hybrid Systems: Computation and Control: 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings*, chapter Safe and Secure Networked Control Systems under Denial-of-Service Attacks, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[All14] Arnold O Allen. *Probability, statistics, and queueing theory.* Academic Press, 2014.

[Alt12] Oliver Alt. *Modellbasierte Systementwicklung mit SysML.* Carl Hanser Verlag GmbH Co KG, 2012.

[AMA17] AMALTHEA4public. Amalthea - an open platform project for embedded multicore systems. `http://www.amalthea-project.org/`, 2017.

[AMY09] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131, Aug 2009.

[Apa17] Apache Software Foundation. Apache thrift software framework. `https://thrift.apache.org/`, 2017.

[AS-17] AS-International Association e.V. As-interface. `http://www.as-interface.net`, 2017.

[AS00] L. B. Arief and N. A. Speirs. A uml tool for an automatic generation of simulation programs. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 71–76, New York, NY, USA, 2000. ACM.

[AUT17] AUTOSAR Foundation. Autosar (automotive open system architecture). `http://www.autosar.org/`, 2017.

[Bas92] Victor R Basili. Software modeling and measurement: the goal/-question/metric paradigm. Technical report, University of Maryland, 1992.

[BB00] Guillem Bernat and Alan Burns. An approach to symbolic worst-case execution time analysis. In *In 25th IFAC Workshop on Real-Time Programming*, 2000.

[BBM13] Matthias Becker, Steffen Becker, and Joachim Meyer. Simulizar: Design-time modeling and performance analysis of self-adaptive systems. *Software Engineering*, 213:71–84, 2013.

[BdMIS04] S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, May 2004.

[Bec08] Steffen Becker. *Coupled model transformations for QoS enabled component-based software design*. PhD thesis, Universitaet Oldenburg, Uhlhornsweg 49-55, 26129 Oldenburg, 2008.

[Bec16a] Beckhoff Automation. Beckhoff Automation Device Specification (ADS). `http://infosys.beckhoff.de/index.php?content=../content/1031/tcadscommon/html/tcadscommon_introads.htm&id=`, 2016. Accessed: 2016-02-19.

[Bec16b] Beckhoff Automation. TF1910 | TC3 UML Beckhoff Automation: Integration of uml (unified modeling language) in twincat 3.1. `http://www.beckhoff.de/default.asp?twincat/tf1910.htm`, 2016. Accessed: 2016-01-15.

[Bec17] Beckhoff Automation. Beckhoff Information System. `http://infosys.beckhoff.de`, 2017. Accessed: 2017-02-19.

[BFF90] Benjamin S Blanchard, Wolter J Fabrycky, and Walter J Fabrycky. *Systems engineering and analysis*, volume 4. Prentice Hall Englewood Cliffs, NJ, 1990.

[BFV08] Reinder J Bril, Gerhard Fohler, and Wim FJ Verhaegh. Execution times and execution jitter analysis of real-time tasks under fixed-priority pre-emptive scheduling. *External Report, Computer Science Report*, page 13, 2008.

[BGMO06] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance prediction of component-based systems: A survey from an engineering perspective. In *ARCHITECTING SYSTEMS WITH TRUSTWORTHY COMPONENTS, VOLUME 3938 OF LNCS*, pages 169–192. Springer, 2006.

[BJN$^+$06] Manfred Broy, Matthias Jarke, Manfred Nagl, Hans Dieter Rombach, Armin B Cremers, Jürgen Ebert, Sabine Glesner, Martin Glinz, Michael Goedicke, Gerhard Goos, et al. Dagstuhl-manifest zur strategischen bedeutung des software engineering in deutschland. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[BKFVH14] Giacomo Barbieri, Konstantin Kernschmidt, Cesare Fantuzzi, and Birgit Vogel-Heuser. A sysml based design pattern for the high-level development of mechatronic systems to enhance re-usability. *IFAC Proceedings Volumes*, 47(3):3431–3437, 2014.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM, 2007.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[BLB13] Matthias Becker, Markus Luckey, and Steffen Becker. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA 13, pages 43–52, New York, NY, USA, 2013. ACM.

[BM94] Alfredo Baginski and Martin Müller. Interbus-s. *Grundlagen und Praxis, Hüthig*, 1994.

[BM03] Simonetta Balsamo and Moreno Marzolla. A simulation-based approach to software performance modeling. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 363–366. ACM, 2003.

[BOF$^+$14] B Beihoff, C Oster, S Friedenthal, C Paredis, D Kemp, H Stoewer, D Nichols, and J Wade. A world in motion–systems engineering vision 2025. *INCOSE-SE Leading Indicators Guide, 2014*, 2014.

[Bon09] Yahor Bondarau. *Design-time performance analysis of component-based real-time systems*. PhD thesis, Citeseer, 2009.

[BR05] Manfred Broy and Andreas Rausch. Das neue v-modell® xt. *Informatik-Spektrum*, 28(3):220–229, 2005.

[BRS95] Jörg Becker, Michael Rosemann, and Reinhard Schütte. Grundsätze ordnungsmäßiger modellierung. *Wirtschaftsinformatik*, 37(5):435–445, 1995.

[CAN17] CAN in Automation (CiA). Can in Automation (CiA). `https://www.can-cia.org/`, 2017.

[CBB15] Robert Cloutier, Clifton Baldwin, and Mary Alice Bone. *Systems Engineering Simplified*. CRC Press, 2015.

[CFA$^+$07] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M.F.P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 185–197, March 2007.

[CHL$^+$03] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[CHO10] Anton Cervin, Dan Henriksson, and Martin Ohlin. Truetime 2.0 beta?reference manual. *Department of Automatic Control, Lund University (June 2010)*, 2010.

[CLP17] CLPA Europe. Cc-link partner association. `http://www.clpa-europe.com/`, 2017.

[COH07] Anton Cervin, Martin Ohlin, and Dan Henriksson. Simulation of networked control systems using truetime. In *Proc. 3rd International Workshop on Networked Control Systems: Tolerant to Faults*, 2007.

[CP03] Murray Cantor and RUP Plug. Rational unified process for systems engineering part 1: Introducing rup se version 2.0. *The Rational Edge (August 2003)*, 2003.

[CPR07] Vittorio Cortellessa, Pierluigi Pierini, and Daniele Rossi. Integrating software models and platform models for performance analysis. *IEEE Transactions on Software Engineering*, 33(6), 2007.

[CS00] Guo Chuanxiong and Zheng Shaoren. Analysis and evaluation of the tcp/ip protocol stack of linux. In *Communication Technology Proceedings, 2000. WCC - ICCT 2000. International Conference on*, volume 1, pages 444–453 vol.1, 2000.

[Das17] Dassault Systemes SolidWorks Corporation. Eplan engineering configuration tool for mechatronic configuration and automated documentation. `http://www.solidworks.de/`, 2017.

[DDGI14] Rafal Dorociak, Roman Dumitrescu, Jürgen Gausemeier, and Peter Iwanek. Specification technique consens for the description of self-optimizing systems. In *Design Methodology for Intelligent Technical Systems*, chapter 4.1, pages 119–127. Springer-Verlag Berlin Heidelberg, January 2014.

[Des00] Philippe Desfray. Uml profiles versus metamodel extensions: An ongoing debate. In *OMG's UML Workshops: UML in the. com Enterprise: Modeling CORBA, Components, XML/XMI and Metadata Workshop*, pages 6–9, 2000.

[dGJKK12] Thijmen de Gooijer, Anton Jansen, Heiko Koziolek, and Anne Koziolek. An industrial case study of performance and cost design space exploration. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 205–216, New York, NY, USA, 2012. ACM.

[DM97] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 68–75, New York, NY, USA, 1997. ACM.

[dMLH$^+$00] Miguel de Miguel, Thomas Lambolais, Mehdi Hannouz, Stéphane Betgé-Brezetz, and Sophie Piekarec. Uml extensions for the specification and evaluation of latency constraints in architectural models. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 83–88, New York, NY, USA, 2000. ACM.

[Dor11] Dov Dori. *Object-process methodology: A holistic systems paradigm*. Springer Science & Business Media, 2011.

[Dou04] Bruce Powel Douglass. *Real time UML: advances in the UML for real-time systems*. Addison-Wesley Professional, 2004.

[Dou06] Bruce Powel Douglass. The harmony process: The transition to software engineering, 2006.

[DPP$^+$16] Stefan Dziwok, Uwe Pohlmann, Goran Piskachev, David Schubert, Sebastian Thiele, and Christopher Gerking. The mechatronicuml design method: Process and language for platform-independent modeling. Technical Report tr-ri-16-352, Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Zukunftsmeile 1, 33102 Paderborn, Germany, December 2016. Version 1.0.

[DSA$^+$14] Peter Danielis, Jan Skodzik, Vlado Altmann, Eike Bjoern Schweissguth, Frank Golatowski, Dirk Timmermann, and Joerg Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8. IEEE, 2014.

[Dub11] A. Dubey. Evaluating software engineering methods in the context of automation applications. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 585–590, July 2011.

[E$^+$07] Jeff A Estefan et al. Survey of model-based systems engineering (mbse) methodologies. *Incose MBSE Focus Group*, 25(8), 2007.

[EBGR01] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, Jul 2001.

[Ecl17a] Eclipse Foundation. Eclipse ide. `https://eclipse.org`, 2017.

[Ecl17b] Eclipse Foundation. Eclipse modeling framework (emf). `https://www.eclipse.org/modeling/emf/`, 2017.

[Ecl17c] Eclipse Foundation. Papyrus modeling environment. `https://eclipse.org/papyrus`, 2017.

[Ecl17d] Eclipse Foundation. Xtend (version 2.11). `http://www.eclipse.org/xtend/`, 2017. Accessed: 2017-02-19.

[EES$^+$02] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4(4):437–455, 2002.

[EFR08] Irene Eusgeld, Felix C. Freiling, and Ralf Reussner, editors. *Dependability Metrics: Advanced Lectures.* Springer-Verlag, Berlin, Heidelberg, 2008.

[EGEE$^+$08] Joseph P Elm, Dennis Goldenson, Khaled El Emam, Nicole Donatelli, Angelica Neisa, NDIA SE Effectiveness Committee, et al. A survey of systems engineering effectiveness-initial results. Technical report, Carnegie Mellon University, 2008.

[EPL17] EPLAN Software & Service GmbH. Eplan engineering configuration tool for mechatronic configuration and automated documentation. `https://www.eplan.de`, 2017.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* Disseration for the degree of doctor of philosophy in computer systems, Uppsala University, June 2003. ISBN 91-554-5671-5 ISSN 1104-2516.

[Eth17a] Ethercat Technology Group and others. Ethercat-der ethernet feldbus. ethercat technology group. https://www.ethercat.org, 2017. Accessed: 2017-02-20.

[Eth17b] Ethernet POWERLINK Standardization Group (EPSG). Ethernet powerlink. `http://www.ethernet-powerlink.org/`, 2017.

[FBH05] Viktoria Firus, Steffen Becker, and Jens Happe. Parametric performance contracts for qml-specified software components. *Electronic Notes in Theoretical Computer Science*, 141(3):73–90, 2005.

[FCF$^+$13] K. Falkner, V. Chiprianov, N.J.G. Falkner, C. Szabo, J. Hill, G. Puddy, D. Fraser, A. Johnston, M. Rieckmann, and A. Wallis. Model-driven performance prediction of distributed real-time embedded defense systems. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 155–158, July 2013.

[FCS12] J. Feljan, J. Carlson, and T. Seceleanu. Towards a model-based approach for allocating tasks to multicore processors. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 117–124, 2012.

[Fee01] Laura Marie Feeney. An energy consumption model for performance analysis of routing protocols for mobile ad hoc networks. *Mobile Networks and Applications*, 6(3):239–249, 2001.

[FEH$^+$12] T. Frank, K. Eckert, T. Hadlich, A. Fay, C. Diedrich, and B. Vogel-Heuser. Workflow and decision support for the design of distributed automation systems. In *IEEE 10th International Conference on Industrial Informatics*, pages 293–299, July 2012.

[FEH$^+$13] Timo Frank, Karin Eckert, Thomas Hadlich, Alexander Fay, Christian Diedrich, and Birgit Vogel-Heuser. Erweiterung des v-modells® für den entwurf von verteilten automatisierungssystemen. *at-Automatisierungstechnik Methoden und Anwendungen der Steuerungs-, Regelungs-und Informationstechnik*, 61(2):79–91, 2013.

[FFMT04] Paolo Ferrari, Alessandra Flammini, Daniele Marioli, and Andrea Taroni. Experimental evaluation of profinet performance. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 331–334. IEEE, 2004.

[FGHW99] Anja Feldmann, Anna C Gilbert, Polly Huang, and Walter Willinger. Dynamics of ip traffic: A study of the role of variability and the impact of control. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 301–313. ACM, 1999.

[FH04] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

[FH12] Jens Frieben and Henning Heutger. Case study : Palladio-based modular system for simulating plc performance. In *Palladio Days 2012*, Karlsruhe Reports in Informatics ; 2012,21 ISSN: 2190-4782, pages 29–37. Karlsruhe Institute of Technology, Karlsruhe, 2012.

[FHL$^+$01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Embedded Software*, pages 469–485. Springer, 2001.

[FHMB13] Jens Frieben, Henning Heutger, Matthias Meyer, and Steffen Becker. Modulare leistungsprognose von kompaktsteuerungen. In *9. Paderborner Workshop Entwurf mechatronischer Systeme*, pages 147–160, Paderborn, April 2013. Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn.

[FK98] Svend Frølund and Jari Koistinen. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5(4):179, 1998.

[FL09] Georg Frey and Liu Liu. Modellierung und simulation vernetzter automatisierungs-und regelungssysteme in modelicamodeling and simulation of networked automation and control systems in modelica. *at-Automatisierungstechnik Methoden und Anwendungen der Steuerungs-, Regelungs-und Informationstechnik*, 57(9):466–476, 2009.

[Fle10] FlexRay Consortium and others. Flexray communications system protocol specification version 3.0.1, 2010.

[FMS04] KF Früh, Uwe Maier, and Dieter Schaudel. *Handbuch der Prozessautomatisierung*. München: Oldenbourg Industrieverlag, 2004.

[FMS08] Sanford Friedenthal, Alan Moore, and Rick Steiner. Systems modeling language (omg sysml) tutorial. In *INCOSE international symposium*, volume 18, pages 1731–1862. Wiley Online Library, 2008.

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[Fra99] Roy Gregory Franks. *Performance analysis of distributed server systems*. PhD thesis, Carleton University Ottawa, 1999.

[Fra06] Ursula Frank. *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme*. PhD thesis, Uni Paderborn, 2006.

[Fre16] FreeRTOS - Cross Platform Real Time Operating System (RTOS). `http://www.freertos.org/`, 2016. Accessed: 2016-01-05.

[FT14] Tanja Frieben and Ansgar Trächtler. Virtual commissioning by means of an adaptive selection of the modeling depth. In *Proceedings of the ASME 2014 International Mechanical Engineering Congress & Exposition IMECE14, Montreal, Quebec, Canada*. ASME, November 14 - 20 2014.

[G$^+$01] Gerhard Gruhler et al. Feldbusse und gerätekommunikationssysteme. *Franzis, Poing*, pages 978–3772357459, 2001.

[GB12] Eva Geisberger and Manfred Broy, editors. *agendaCPS: Integrierte Forschungsagenda Cyber-Physical Systems*. acatech Studie. Springer, Berlin, 2012.

[GCD$^+$14] Jürgen Gausemeier, Anja Maria Czaja, Roman Dumitrescu, Christian Tschirner, Daniel Steffen, and Olga Wiederkehr. Studie: Systems engineering in der industriellen praxis. Broschüre, January 2014.

[GDKN11] J. Gausemeier, R. Dumitrescu, S. Kahl, and D. Nordsiek. Integrative development of product and production system for mechatronic products. *Robotics and Computer-Integrated Manufacturing*, 27(4):772 – 778, 2011. Conference papers of Flexible Automation and Intelligent ManufacturingIntelligent manufacturing and services.

[Gee05] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[GFDK09] J. Gausemeier, U. Frank, J. Donoth, and S. Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20(4):201, 2009.

[GG13] Hans-Jürgen Gevatter and Ulrich Grünhaupt. *Handbuch der Mess- und Automatisierungstechnik in der Produktion*. Springer-Verlag, 2013.

[GGS$^+$07] Jürgen Gausemeier, Holger Giese, Wilhelm Schäfer, Björn Axenath, Ursula Frank, Stefan Henkler, Sebastian Pook, and Matthias Tichy. Towards the design of self-optimizing mechatronic systems: Consistency between domain-spanning and domain-specific models. In *International Conference On Engineering Design, ICED'07*, 28-31 August and Paris and France, 2007.

[GH01] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, 2001.

[GH09] V. C. Gungor and G. P. Hancke. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *IEEE Transactions on Industrial Electronics*, 56(10):4258–4265, Oct 2009.

[GH13] B. Galloway and G.P. Hancke. Introduction to industrial control networks. *Communications Surveys Tutorials, IEEE*, 15(2):860–880, Second 2013.

[GJF13] P. Gaj, J. Jasperneite, and M. Felser. Computer communication within industrial distributed environment - a survey. *Industrial Informatics, IEEE Transactions on*, 9(1):182–189, Feb 2013.

[GLL12] J. Gausemeier, G. Lanza, and U. Lindemann. *Produkte und Produktionssysteme integrativ konzipieren - Modellbildung und Analyse in der frühen Phase der Produktentstehung*. Hanser Verlag, München, 2012.

[GM04] Vincenzo Grassi and Raffaela Mirandola. Towards automatic compositional performance analysis of component-based systems. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, pages 59–63, New York, NY, USA, 2004. ACM.

[GMM97] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 317–324, New York, NY, USA, 1997. ACM.

[GRS14] Jürgen Gausemeier, Franz Josef Rammig, and Wilhelm Schäfer. Design methodology for intelligent technical systems. *Lecture Notes in Mechanical Engineering. Springer*, 1(2):3, 2014.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 19, pages 120–132. ACM, 1991.

[Han13] Yiming Han. Application performance evaluation on different compiler optimizations. *Advances in Computer Science and its Applications*, 2(3):410–415, 2013.

[Hap04] Jens Happe. Reliability Prediction of Component-Based Software Architectures. Master's thesis, University of Oldenburg, 2004.

[Hap05] Jens Happe. Performance Prediction for Embedded Systems, 2005.

[Hap08] Jens Happe. *Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments.* Dissertation, University of Oldenburg, Germany, August 2008.

[Hap16] Jens Happe. Exact Schedulers Palladio Addon. `https://sdqweb.ipd.kit.edu/wiki/Exact_Schedulers`, 2016. Accessed: 2016-12-02.

[HBM$^+$15] Jörg Holtmann, Ruslan Bernijazov, Matthias Meyer, David Schmelter, and Christian Tschirner. Integrated systems engineering and software requirements engineering for technical systems. In *Proceedings of the International Conference on Software and Systems Process (ICSSP)*, pages 57–66, New York, NY, USA, August 2015. ACM. Best full paper ICSSP 2015.

[HBM$^+$16] Jörg Holtmann, Ruslan Bernijazov, Matthias Meyer, David Schmelter, and Christian Tschirner. Integrated and iterative systems engineering and software requirements engineering for technical systems. *Journal of Software Evolution and Process*, May 2016.

[HCÅ03] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. Truetime: Real-time control system simulation with matlab/simulink. In *Proceedings of the Nordic Matlab conference*, 2003.

[HH08] Reinhard Höhn and Stephan Höppner. *Das V-Modell XT: Grundlagen, Methodik und Anwendungen.* Springer-Verlag, 2008.

[HHM09] Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. Statistical-based wcet estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[Hil05] Jane Hillston. *A compositional approach to performance modelling*, volume 12. Cambridge University Press, 2005.

[Hil09] Jane Hillston. University of edinburg - handout: Teaching course performance modelling. `http://www.inf.ed.ac.uk/teaching/courses/pm/handouts/stochasticpetrinets.pdf`, 2009. Accessed: 2017-08-03.

[HJ03] Mahbub Hassan and Raj Jain. *High performance TCP/IP networking*, volume 29. Prentice Hall, 2003.

[HKW05] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 180–188, New York, NY, USA, 2005. ACM.

[HM08] Wolfgang Hesse and Heinrich C Mayr. Modellierung in der softwaretechnik: eine bestandsaufnahme. *Informatik-Spektrum*, 31(5):377–393, 2008.

[HMR13] Jörg Henss, Philipp Merkle, and Ralf H Reussner. The ompcm simulator for model-based software performance prediction: poster abstract. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pages 354–357. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[Hof08] Hans-Peter Hoffmann. Harmony/se: A sysml based systems engineering process. *Innovation*, 2008.

[HPV15] F Herrera, P Peñil, and E Villar. Uml/marte network modelling methodology. Technical report, University of Cantabria, 2015.

[HSST13] Christian Heinzemann, Oliver Sudmann, Wilhelm Schäfer, and Matthias Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proceedings of the 2013 International Conference on Software and System Process*, ICSSP 2013, pages 36–45. ACM, New York, NY, USA, 18 - 19 May 2013.

[HW00] Luc Hohwiller and Serge Wendling. Fieldbus network simulation using a time extended estelle formalism. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 92–97. IEEE, 2000.

[IH08] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[IKDN13] Peter Iwanek, Lydia Kaiser, Roman Dumitrescu, and Alexander Nyßen. Fachdisziplinübergreifende Systemmodellierung mechatronischer Systeme mit SysML und CONSENS. In *Tag des Systems Engineering 2013*, 2013.

[INC17] INCHRON. Inchron tool-suite. `https://www.inchron.com`, 2017.

[Int03] International Electrotechnical Commission (IEC). IEC 61158: Digital data communications for measurement and control-fieldbus for use in industrial control systems, 2003.

[Int13a] International Electrotechnical Commission (IEC). IEC 61131-3: Programmable controllers - Part 3: Programming languages, 2013.

[Int13b] International Electrotechnical Commission (IEC). IEC 62264: Enterprise-control system integration - part 1: Models and terminology. `https://webstore.iec.ch/publication/6675`, 2013.

[Int16a] International Electrotechnical Commission (IEC). `http://www.iec.ch`, 2016. Accessed: 2016-07-08.

[Int16b] International Electrotechnical Commission (IEC). IEC 61158 (Digital data communication for measurement and control - Fieldbus for use in industrial control systems) . `https://webstore.iec.ch/home`, 2016. Accessed: 2016-02-19.

[Ips17] Ipswitch, Inc. Whatsup gold. `http://www.whatsupgold.com`, 2017.

[ISG17] ISG Industrielle Steuerungstechnik GmbH. Isg-virtuos. `http://www.isg-stuttgart.de/`, 2017.

[ITE17] ITEA Trust4All Consortium. CARAT-RTIE Performance toolkit. `http://www.win.tue.nl/trust4all/`, 2017. Accessed: 2017-01-13.

[Jai90] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[JHHF09] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *2009 IEEE International Conference on Software Maintenance*, pages 125–134, Sept 2009.

[JLS11] Hongchao Ji, Oliver Lenord, and Dieter Schramm. A model driven approach for requirements engineering of industrial automation systems. In *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools;*

Zurich; Switzerland; September 5; 2011, number 56, pages 9–18. Linkø"ping University Electronic Press; Linkø"pings universitet, 2011.

[JT10] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids.* Springer Science & Business Media, 2010.

[Kag12] Henning Kagermann. *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0.* Forschungsunion im Stifterverband für die Deutsche Wirtschaft e.V, Berlin, 2012.

[Kah01] Pekka Kahkipuro. Uml-based performance modeling framework for component-based distributed systems. In *Performance Engineering, State of the Art and Current Trends*, pages 167–184, London, UK, UK, 2001. Springer-Verlag.

[KDHM13] Lydia Kaiser, Roman Dumitrescu, Jörg Holtmann, and Matthias Meyer. Automatic verification of modeling rules in systems engineering for mechatronic systems. In *Proceedings of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference.* ASME, ASME, July 2013.

[KG08] Samuel Kounev and Ian Gorton. *Performance Evaluation: Metrics, Models and Benchmarks: SPEC International Performance Evaluation Workshop, SIPEW 2008, Darmstadt, Germany, June 27-28, 2008, Proceedings*, volume 5119. Springer, 2008.

[KG17] SAP Deutschland SE & Co. KG. SAP S/4HANA. `https://www.sap.com/germany/product/enterprise-management`, 2017.

[KHCD17] Nafiseh Kahani, Nicolas Hili, James R. Cordy, and Juergen Dingel. Evaluation of uml-rt and papyrus-rt for modelling self-adaptive systems. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering*, MISE '17, pages 12–18, Piscataway, NJ, USA, 2017. IEEE Press.

[KLL03] Alexander Klemm, Christoph Lindemann, and Marco Lohmann. Modeling ip traffic using the batch markovian arrival process. *Performance Evaluation*, 54(2):149–173, 2003.

[Kol60] Andrey N. Kolmogorov. *Foundations of the Theory of Probability.* Chelsea Pub Co, 2 edition, June 1960.

[Koy13] Evelina Koycheva. *Entwurfsbegleitende Leistungsanalyse mit UML, MARTE und Generalisierten Netzen.* Walter de Gruyter, 2013.

[KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug 2002.

[KRK95] W. Kriesel, H. Rohr, and A. Koch. *Geschichte und Zukunft der Mess- und Automatisierungstechnik*. Technikgeschichte in Einzeldarstellungen. VDI Verlag, 1995.

[KVH13] K. Kernschmidt and B. Vogel-Heuser. An interdisciplinary sysml based modeling approach for analyzing change influences in production plants to support the engineering. In *2013 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1113–1118, Aug 2013.

[LaÎ99] Thierry LaÎnÉ. *Fieldbus Technology: Systems Integration, Networking, and Engineering Proceedings of the Fieldbus Conference FeT'99 in Magdeburg, Federal Republic of Germany, September 23-24,1999*, chapter Internet Technologies and Fieldbuses, pages 61–68. Springer Vienna, Vienna, 1999.

[Lau13] Rudolf Lauber. *Prozeßautomatisierung I: Aufbau und Programmierung von Prozeßrechensystemen*. Springer-Verlag, 2013.

[Leh12] Sebastian Lehrig. Assessing the quality of model-to-model transformations based on scenarios. Master's thesis, University of Paderborn, Zukunftsmeile 1, October 2012.

[Leu04] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[LF07] Liu Liu and G. Frey. Simulation approach for evaluating response times in networked automation systems. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 1061–1068, 2007.

[LF12] Liu Liu and Georg Frey. Effiziente modellierung und simulation von kommunikationsnetzen in modelica. In Wolfgang A. Halang, editor, *Kommunikation unter Echtzeitbedingungen*, Informatik Aktuell, pages 89–98. Springer, 2012.

[LFM00] Howard Lykins, Sanford Friedenthal, and Abraham Meilich. 4.4.4 adapting uml for an object oriented systems engineering method (oosem). *INCOSE International Symposium*, 10(1):490–497, 2000.

[LFVH13] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pages 4386–4391, Nov 2013.

[Lin16] Real-Time Linux Wiki config_preempt_rt patch for linux. `https://rt.wiki.kernel.org/index.php/Main_Page`, 2016. Accessed: 2016-01-05.

[LL99] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66 – 104, 1999.

[LL02] Kyung Chang Lee and Suk Lee. Performance evaluation of switched ethernet for real-time industrial communications. *Computer standards & interfaces*, 24(5):411–423, 2002.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007. `http://www.comp.nus.edu.sg/~rpembed/chronos`.

[LM12] Yau-Tsun Steven Li and Sharad Malik. *Performance analysis of real-time embedded software*. Springer Science & Business Media, 2012.

[LMT99] F.-L. Lian, J. R. Moyne, and D. M. Tilbury. Performance evaluation of control networks for manufacturing systems. In *In Proceedings of the ASME International Mechanical Engineering Congress and Exposition (Dynamic Systems and Control Division*, pages 6–7, 1999.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3):257–279, July 1999.

[Lop15] Margaret L. Loper. *Modeling and Simulation in the Systems Engineering Life Cycle*. Springer-Verlag London, 2015.

[LPT10] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: a hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.

[Lun08] Jan Lunze. *Automatisierungstechnik: Methoden für die Überwachung und Steuerung kontinuierlicher und ereignisdiskreter Systeme*. Oldenbourg Verlag, 2008.

[LWF08] L Liu, F Wagner, and G Frey. Simulation verteilter automatisierungssysteme in modelica. *VDIBERICHT*, 2032:185, 2008.

[LWH05] M. Long, Chwan-Hwa John Wu, and J.Y. Hung. Denial of service attacks on network-based control systems: impact and mitigation. *Industrial Informatics, IEEE Transactions on*, 1(2):85–96, May 2005.

[Mah97] Bruce A Mah. An empirical model of http network traffic. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 2, pages 592–600. IEEE, 1997.

[Mah13] Nitaigour P Mahalik. *Fieldbus technology: industrial network standards for real-time distributed control.* Springer Science & Business Media, 2013.

[Man00] Charles C Mann. The end of moores law. *Technology Review*, 103(3):42–48, 2000.

[Mat16] MathWorks. MATLAB (matrix laboratory) numerical computing environment. http://www.mathworks.de/products/matlab/, 2016. Accessed: 2016-07-12.

[Mat17] MathWorks. Simulink - simulation and model-based design. `https://de.mathworks.com/products/simulink.html`, 2017.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 75–84, New York, NY, USA, 1991. ACM.

[MBB$^+$89] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. The effect of execution policies on the semantics and analysis of stochastic petri nets. *IEEE Transactions on Software Engineering*, 15(7):832–846, Jul 1989.

[MBC$^+$94] Marco Ajmone Marsan, G. Balbo, Gianni Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[MDFF06a] G. Marsal, B. Denis, J. M. Faure, and G. Frey. Evaluation of response time in ethernet-based automation systems. In *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 380–387, Sept 2006.

[MDFF06b] Gaëlle Marsal, Bruno Denis, J-M Faure, and Georg Frey. Evaluation of response time in ethernet-based automation systems. In *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*, pages 380–387. IEEE, 2006.

[Men16] Mentor Graphics. Nucleus©RTOS mentor graphics real time operating system. `https://www.mentor.com/embedded-software/nucleus/`, 2016. Accessed: 2016-01-05.

[Mic16] Microsoft. Windows Embedded windows embedded family of operating systems. `http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx`, 2016. Accessed: 2016-01-05.

[Mil89] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[MLK06] Pulat Matkurbanov, SeungKi Lee, and Dong-Sung Kim. A survey and analysis of wireless fieldbus for industrial environments. In *SICE-ICASE, 2006. International Joint Conference*, pages 5555–5561, Oct 2006.

[Mod04] IDA Modbus. Modbus application protocol specification v1. 1a. *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2004.

[Mod17a] Modbus organization. Modbus organization. `http://www.modbus.org`, 2017.

[Mod17b] Modelica Association. Modelica. `https://www.modelica.org/`, 2017. Accessed: 2016-06-12.

[Mol82] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, Sept 1982.

[MR04] Perry S Marshall and John S Rinaldi. *Industrial Ethernet.* ISA, 2004.

[MSUW02] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-driven architecture. In *International Conference on Object-Oriented Information Systems*, pages 290–297. Springer, 2002.

[MU09] Leon McGinnis and Volkan Ustun. A simple example of sysml-driven simulation. In *Winter Simulation Conference*, WSC '09, pages 1703–1710. Winter Simulation Conference, 2009.

[MWD$^+$05] G Marsal, D Witsch, B Denis, JM Faure, and G Frey. Evaluation of real-time capabilities of ethernet-based automation systems using formal verification and simulation. *Proceedings of RJCITR*, 5:27–30, 2005.

[MZCW04] Alexander Maxiaguine, Yongxin Zhu, Samarjit Chakraborty, and Weng-Fai Wong. Tuning soc platforms for multimedia processing: Identifying limits and tradeoffs. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 128–133. ACM, 2004.

[NS-17] NS-3 Consortium. NS-3 discrete-event network simulator. `https://www.nsnam.org/`, 2017. Accessed: 2017-02-15.

[Obj04] Object Management Group (OMG). Object Management Group – Data Distribution Service (DDS), Version 1.4 . `http://www.omg.org/spec/DDS/`, 2004. Accessed: 2016-02-19.

[Obj05] Object Management Group (OMG). OMG uml profile for Schedulability, Performance, and Time (SPTP). version 1.1, 2005.

[Obj06] Object Management Group (OMG). OMG the uml profile for marte: Modeling and Analysis of Real-Time and Embedded systems. version 1.1. http://www.omgmarte.org/, 2006.

[Obj11a] Object Management Group (OMG). OMG Business Process Model and Notation (BPMN) version 2.0. *OMG Specification, Object Management Group*, 2011.

[Obj11b] Object Management Group (OMG). OMG Meta Object Facility (MOF) 2.0 query/view/transformation. *OMG Specification, Object Management Group*, 2011.

[Obj15a] Object Management Group (OMG). Object Management Group - Unified Modeling Language version 2.5. `http://www.omg.org/spec/UML/2.5/`, 2015. Accessed: 2016-03-07.

[Obj15b] Object Management Group (OMG). OMG Systems Modeling Language (omg sysml) v1.4. http://www.omgsysml.org/, 2015.

[Obj17] Object Management Group (OMG). OMG Object Constrait Language (OCL). `http://www.omg.org/spec/OCL/`, 2017.

[ODV17] ODVA Inc. Odva organization. `https://www.odva.org/`, 2017.

[Ope17] OpenSim Ltd. Omnet++ - discrete event simulator. `https://omnetpp.org/`, 2017.

[Ora17] Oracle. Oracle e-business suite. `http://www.oracle.com/us/products/applications/ebusiness/overview/index.html`, 2017.

[Org17] Organization for the Advancement of Structured Information Standards (OASIS). Message Queue Telemetry Transport (MQTT)). `http://mqtt.org/`, 2017.

[OS97] Greger Ottosson and Mikael Sjodin. Worst-case execution time analysis for modern hardware architectures. In *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97*. Citeseer, 1997.

[Par12] Dominique Paret. *FlexRay and its applications: real time multiplexed network*. John Wiley & Sons, 2012.

[PB00] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

[Per06] Simon Perathoner. *Evaluation and Comparison of Performance Analysis Methods for Distributed Embedded Systems*. PhD thesis, Politecnico di Milano, Politecnico di Milano, 2006.

[Pet83] James Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice Hall, 1983.

[Pho16] Phoenix Contact. Phoenix Contact – ProConOS embedded CLR. `https://www.phoenixcontact-software.com/de/iec-61131-control/laufzeitsysteme/proconos-eclr`, 2016. Accessed: 2016-02-19.

[PJ08] Christiaan JJ Paredis and Thomas Johnson. Using omgs sysml to support simulation. In *Simulation Conference, 2008. WSC 2008. Winter*, pages 2350–2352. IEEE, 2008.

[PKC96] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Network Protocols, 1996. Proceedings., 1996 International Conference on*, pages 171–180. IEEE, 1996.

[PMDB14] Uwe Pohlmann, Matthias Meyer, Andreas Dann, and Christopher Brink. Viewpoints and views in hardware platform modeling for safe deployment. In *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '14, pages 23:23–23:30, New York, NY, USA, 2014. ACM.

[PN09] Carlos E. Pereira and Peter Neumann. *Industrial Communication Protocols*, pages 981–999. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Pro16a] Profibus and Profinet International (PI). `http://www.profibus.com`, 2016. Accessed: 2016-02-19.

[Pro16b] Profibus Nutzerorganisation e.V. (PNO). `http://www.interbusclub.com/`, 2016. Accessed: 2016-02-19.

[Pro17] Profibus International (PI). Profibus standard iec 61158. `www.profibus.com`, 2017.

[Pry08] Gunnar Prytz. A performance analysis of ethercat and profinet irt. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 408–415. IEEE, 2008.

[PS85] James Lyle Peterson and Abraham Silberschatz. *Operating system concepts*, volume 2. Addison-Wesley Reading, MA, 1985.

[PS01] Frederick M Proctor and William P Shackleford. Real-time operating system timing jitter and its impact on motor control. In *Intelligent Systems and Advanced Manufacturing*, pages 10–16. International Society for Optics and Photonics, 2001.

[QNX16] QNX. QNX Neutrino RTOS the qnx neutrino rtos (realtime operating system). `http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html`, 2016. Accessed: 2016-01-05.

[RBB$^+$11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, Universität Karlsruhe (TH), Karlsruhe, 2011.

[RBG13] V. Rudtsch, F. Bauer, and J. Gausemeier. Approach for the conceptual design validation of production systems using automated simulation-model generation. *Procedia Computer Science*, 16:69 – 78, 2013.

[RBH$^+$95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. *SIGOPS Oper. Syst. Rev.*, 29(5):285–298, December 1995.

[RBK$^+$07] Ralf H. Reussner, Steffen Becker, Heiko Koziolek, Jens Happe, Michael Kuperberg, and Klaus Krogmann. The Palladio Component Model. Interner Bericht 2007-21, Universität Karlsruhe (TH), 2007. October 2007.

[Rec08] Jörg Rech. *Model-Driven Software Development: Integrating Quality Assurance: Integrating Quality Assurance*. IGI Global, 2008.

[RH10] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

[Rie14] Jan Rieke. *Model Consistency Management for Systems Engineering*. Phd thesis, University of Paderborn, July 2014.

[Riv17] Riverbed Technology. Opnet (steelcentral). `https://www.riverbed.com/de/products/steelcentral`, 2017.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[Ros08] Martin Rostan. Industrial ethernet technologies: Overview. In *ETG Industrial Ethernet Seminar Series, Nuremberg*, 2008.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and F Plasil. The common component modeling example. *Lecture notes in computer science*, 5153, 2008.

[RS94] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, Jan 1994.

[RTA16] Real-Time Linux RTAI rtai - real time application interface official website. `https://www.rtai.org/`, 2016. Accessed: 2016-01-05.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[Sch95] Andy Schürr. Specification of graph translators with triple graph grammars. In *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D.* Springer, 1995.

[SEG] SEGGER. embOS Real Time Operating System. `https://www.segger.com/embos.html`. Accessed: 2016-01-05.

[Sel98] Bran Selic. Using uml for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.

[Ser17] Sercos International e.V. Sercos international e.v. `www.sercos.com`, 2017.

[SG13] Bran Selic and Sbastien Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[SGGS98] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[She96] Sarah A Sheard. Twelve systems engineering roles. In *INCOSE International Symposium*, volume 6, pages 478–485. Wiley Online Library, 1996.

[SHK98] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73, August 1998.

[SIEa] SIEMENS. SIMATIC S7 CFC simatic s7 cfc (continuous function chart). `http://w3.siemens.com/mcms/simatic-controller-software/en/programming-options/simatic-s7-cfc/pages/default.aspx`. Accessed: 2016-01-15.

[SIEb] SIEMENS. SIMATIC S7-GRAPH graphical language for describing procedures with alternative or parallel step sequences. `http://w3.siemens.com/mcms/simatic-controller-software/de/step7/simatic-s7-graph/Seiten/Default.aspx`. Accessed: 2016-01-15.

[SIEc] SIEMENS. SIMATIC S7-SCL programming language for complex algorithms, arithmetic functions or for data processing tasks. `http://w3.siemens.com/mcms/simatic-controller-software/de/step7/simatic-s7-scl/Seiten/Default.aspx`. Accessed: 2016-01-15.

[SJ08] Robert Shaw and Brendan Jackman. An introduction to flexray as an industrial network. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1849–1854. IEEE, 2008.

[SKKS11] Till Steinbach, Hermand Dieumo Kenfack, Franz Korf, and Thomas C Schmidt. An extension of the omnet++ inet framework for simulating real-time ethernet with high accuracy. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 375–382. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[SL05] J. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2005.

[Smaa] Smart Software Solutions GmbH (3S). 3S-Smart Software Solutions GmbH codesys runtime. `https://www.codesys.com/`. Accessed: 2016-01-05.

[Smab] Smart Software Solutions GmbH (3S). CODESYS UML codesys professional developer edition for modeling uml class and state diagrams. `http://store.codesys.com/codesys-uml.html`. Accessed: 2016-01-15.

[Smi62] E. C. Smith. Simulation in systems engineering. *IBM Systems Journal*, 1(1):33–50, 1962.

[SN99] Weiming Shen and Douglas H Norrie. Agent-based systems for intelligent manufacturing: a state-of-the-art survey. *Knowledge and information systems*, 1(2):129–156, 1999.

[Spa17] Sparx Systems. Enterprise architect. `http://www.sparxsystems.de/`, 2017.

[SQL] Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). `http://www.iso.org/iso/catalogue_detail.htm?csnumber=53682`. Accessed: 2016-02-19.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer, Wien, New York, 1973.

[Ste09] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling.* Princeton University Press, 2009.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[SVE07] Thomas Stahl, Markus Völter, and Sven Efftinge. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* Dpunkt Verlag, 2007.

[SW06] Miroslaw Staron and Claes Wohlin. An industrial case study on the choice between language customization mechanisms. In *International Conference on Product Focused Software Process Improvement*, pages 177–191. Springer, 2006.

[SW08] Gerhard Schnell and Bernhard Wiedemann. Bussysteme in der automatisierungs-und prozesstechnik. *Vieweg+ Teubner, Wiesbaden*, 2008.

[SW09] Daniel Schütz and Andreas Wannagat. Domänenspezifische modellierung für automatisierungstechnische anlagen mit hilfe der sysml. *atp edition*, 51:54–62, 2009.

[TA17] Timing-Architects. Timing-architects - development-tools for embedded multi-core systems. `https://www.timing-architects.com/`, 2017.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[TDBG15] C. Tschirner, R. Dumitrescu, M. Bansmann, and J. Gausemeier. Tailoring model-based systems engineering concepts for industrial application. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 69–76, April 2015.

[TF11] Kleanthis Thramboulidis and Georg Frey. An mdd process for iec 61131-based industrial automation systems. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE, 2011.

[Tho05] J.-P. Thomesse. Fieldbus technology in industrial automation. *Proceedings of the IEEE*, 93(6):1073–1101, 2005.

[Tid17] Tidorum Ltd. Bound-t - execution time analyzer from tidorum. `http://www.bound-t.com/`, 2017.

[TP09] Lothar Thiele and Simon Perathoner. Performance prediction of distributed platforms. In Gabriela Nicolescu and Pieter J Mosterman, editors, *Model-Based Design of Heterogeneous Embedded Systems*, Computational Analysis, Synthesis, and Design of Dynamic Systems, chapter 1, pages 3–25. CRC Press, November 2009.

[TWTT87] Andrew S Tanenbaum, Albert S Woodhull, Andrew S Tanenbaum, and Andrew S Tanenbaum. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[V^+01] András Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM?2001)*, volume 9, page 65. sn, 2001.

[VDI] VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik. Cyber-physical systems -chancen und nutzen aus sicht der automation.

[Ver04] Verein Deutscher Ingenieure (VDI). 2206: Entwicklungsmethodik für mechatronische systeme. *VDI-Verlag, Düsseldorf*, 2004.

[Ver12] Verband Deutscher Maschinen- und Anlagenbau. Trendstudie: It und automation in den produkten des maschinenbau bis 2015, 2012.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[VHBKF11] Birgit Vogel-Heuser, Steven Braun, Benjamin Kormann, and David Friedrich. Implementation and evaluation of {UML} as modeling notation in object oriented software engineering for machine and plant automation. *{IFAC} Proceedings Volumes*, 44(1):9151 – 9157, 2011. 18th {IFAC} World Congress.

[VHSFL14] Birgit Vogel-Heuser, Daniel Schütz, Timo Frank, and Christoph Legat. Model-driven engineering of manufacturing automation software projects ? a sysml-based approach. *Mechatronics*, 24(7):883 – 897, 2014. 1. Model-Based Mechatronic System Design 2. Model Based Engineering.

[VIN17] VINT Project. NS-2 - Network Simulator. `http://www.isi.edu/nsnam/ns/`, 2017. Accessed: 2017-04-06.

[Vya13] V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, Aug 2013.

[Wan06] Ernesto Wandeler. *Modular performance analysis and interface based design for embedded real time systems*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2006.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[Wei11] Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[Wei15] T. Weilkiens. *Sysmod - The Systems Modeling Toolbox - Pragmatic Mbse with Sysml*. You Lulu Incorporated, 2015.

[Wer09] B. Werner. Object-oriented extensions for IEC 61131-3. *IEEE Industrial Electronics Magazine*, 3(4):36–39, Dec 2009.

[Win] Windriver . VxWorks vxworks - windriver. `http://windriver.com/products/vxworks`. Accessed: 2016-01-05.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[WRF$^+$15] David D. Walden, Garry J. Roedler, Kevin Forsberg, R. Douglas Hamelin, and Thomas M. Shortell, editors. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities.* Wiley, Hoboken, NJ, 4 edition, 2015.

[WS02] Lloyd G Williams and Connie U Smith. Pasa sm: a method for the performance assessment of software architectures. In *Proceedings of the 3rd International Workshop on Software and Performance*, pages 179–189. ACM, 2002.

[WSR99] K. Weiss, T. Steckstor, and W. Rosenstiel. Performance analysis of a rtos by emulation of an embedded system. In *Rapid System Prototyping, 1999. IEEE International Workshop on*, pages 146–151, Jul 1999.

[ZBN93] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.

[ZPK00] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, 2000.

List of Figures

List of Tables

APPENDIX A

Turbocharger example

For this thesis, a medium sized production system provided by ELHA[1] to mill the body of Turbochargers is used as an ongoing example. To protect the intellectual property of ELHA, the provided System Engineering models have been modified to obfuscate the real system structure and its components.

Figure A.1 shows a photo of a similar system with two Mills. The input and output workpieces are transported by robots in-between. Several other components like stations to measure, buffer, or to engrave the workpieces are also part of this production system.

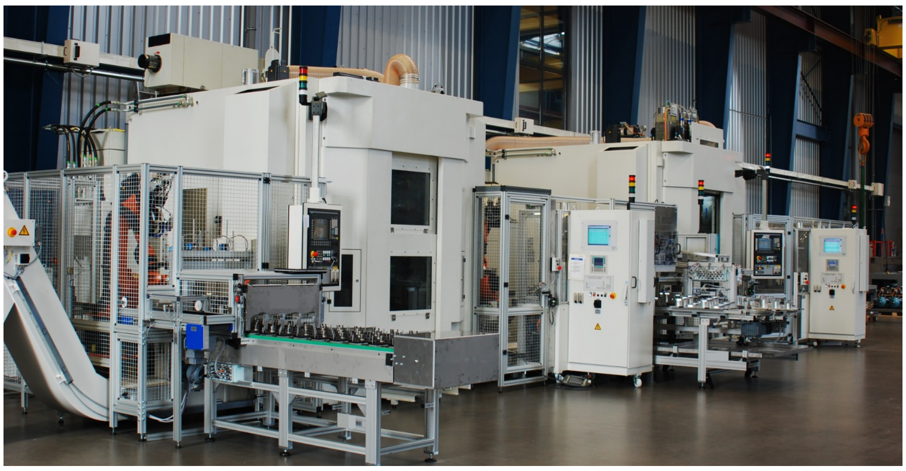


Figure A.1: Photo of two milling centers with automation hardware for workpiece transport (source ELHA)

A.1 Changes to the original turbocharger automation system

The Systems Engineering models provided by ELHA are detailed and contain a wide range of components, subcomponents and flows. For this thesis, the models have been manually transformed from Microsoft Visio documents to

[1]ELHA Maschinenbau – http://www.elha.de/

SysML models created with Papyrus. Some changes have been made to simplify the model and to obfuscate ELHA specific details. The most important changes are as follows

- The number of Function Blocks have been reduced. Similar Function Blocks are merged into single FB. The original software contains more than 370 Function Blocks, hierarchically structured and contained in round about 35 entry level Function Blocks.
- ELHA uses Siemens PLCs to control their machines. This PLC has been exchanged with a Phoenix Contact PLC to be able to gather extensive profiling data for the simulation models and to later compare a real PLC with the simulation data.
- The ELHA Turbocharger mill uses a set of different Fieldbusses including Profibus. This fieldbus is not available for the available Phoenix Contact PLC and had to be replaced with Interbus.
- Several active structure elements and flows have been renamed to obfuscate the details of the production system. This has been done to protect the intellectual property of ELHA.

A.2 List of System- and Environment elements

The following Table A.2 lists all environment and system elements created in the Papyrus SysML model. Each element is identified by an id and provides a short description of its function or use. The coloumn 'PartOf' denotes the id of which this element is a part of.

Table A.1: List of used System and Environment elements

id	Element	PartOf	Description
1	System	–	Root element for the model. Contains also all environment elements
2	Unit1	1	System containing the first mill
3	Unit2	1	System containing the second mill
4	Transport	1	System element for transport, buffering and additional tasks
5	Material-Supply	1	External system that provides workpieces at the beginning of the assembly line
6	SCADA	1	SCADA system for collecting information about the status. Also gathers measurement and engravement information
7	Operator	1	Human operating the production system
8	Workpiece	1	Workpieces in different states that will be processed by the automation system
9	Remote-Diagnostic	1	Additional system for remote diagnosis
10	Maintenance	1	Personal for monitoring, maintenance and repairs

Table A.1: List of used System and Environment elements

id	Element	PartOf	Description
11	Environment	1	Element for abstracting environment influences like vibration onto the system
12	El.Energy-Supply	1	Supply unit providing electrical energy
13	Chippings-Disposal	1	Supply unit for removing and storing chippings
14	CompressedAir	1	Supply unit providing compressed air
15	Oil	1	Supply unit providing oil
16	Cooling	1	Supply unit providing additional cooling liquid
17	Suction	1	Supply unit to clean air
18	AssemblyLine	1	Connection to the following production steps
19	RT-Robot	2	First robot (“Rohteil”) taking parts from the delivery and puts them on the input shuttle
20	PLC	2	PLC controlling the unit 1
21	HMI	2	HMI for accessing data of and controlling unit 1
22	Workpiece-Delivery	2	Workpieces are delivered to the production system by this system
23	Mill	2	The mill processing the workpieces (drill, mill, cut)
24	ShuttleInput	2	System for injecting the workpieces into the mill
25	ShuttleOutput	2	System for ejecting the workpieces out of the mill
26	Rejectslide	2	Slide for damaged workpieces
27	Measurement-Station	4	Station for measuring the workpiece after the processing step. Data is exchanged with SCADA system
28	HFT-Robot	4	“HalbFertigTeil”-Robot for moving workpieces between the different station in the transport unit
29	Supply-Distribution	4	System for distributing the different supplies to the unit1, unit2 and transport parts
30	SPC-Station	4	Station for manual statistical process control
31	BufferStation	4	A station for buffering workpieces in case the unit2 mill is blocked
32	Engravement-Station	4	Workpieces are engraved in this station. Engravement data is send to SCADA system
33	Rejectslide	4	Slide for damaged workpieces

Table A.1: List of used System and Environment elements

id	Element	PartOf	Description
34	PLC	4	PLC controlling the transport unit
35	HMI	4	HMI for accessing data of and controlling the transport unit
36	SPC-Station	4	Station for manual statistical process control
37	Engravement-Station	4	Workpieces are engraved in this station. Engravement data is send to SCADA system
38	Assembly-Connector	4	Delivery system for workpieces to the assembly line for the following production steps
39	Rejectslide	4	Slide for damaged workpieces
40	Mill	3	The mill processing the workpieces (drill, mill, cut)
41	HMI	3	HMI for accessing data of and controlling unit 2
42	FT-Robot	3	"Fertigteil"-Robot for moving workpieces between the different stations in the unit 2
43	PLC	3	PLC controlling the unit 2
44	ShuttleOutput	3	System for ejecting the workpieces out of the mill
45	ShuttleInput	3	System for injecting the workpieces into the mill
46	Measurement-Station	3	Station for measuring the workpiece after the processing step
47	SensorLane1	38	Digital sensor on lane one of the workpiece carrier to check empty position
48	SensorLane2	38	Digital sensor on lane two of the workpiece carrier to check empty position
49	SensorLane3	38	Digital sensor on lane three of the workpiece carrier to check empty position
50	FixingL1Left	38	Left side actuator for fixing/securing the workpiece on the carrier on lane one
51	FixingL1Right	38	Right side actuator for fixing/securing the workpiece on the carrier on lane one
52	FixingL2Left	38	Left side actuator for fixing/securing the workpiece on the carrier on lane two
53	FixingL2Right	38	Right side actuator for fixing/securing the workpiece on the carrier on lane two
54	FixingL3Left	38	Left side actuator for fixing/securing the workpiece on the carrier on lane three
55	FixingL3Right	38	Right side actuator for fixing/securing the workpiece on the carrier on lane three

Table A.1: List of used System and Environment elements

id	Element	PartOf	Description
56	Stopper	38	Actuator for stopping an incoming work-piece carrier
57	WPCRaiser	38	Actuator for lifting/raising the workpiece from the transport lane to reduce vibrati-ons
58	EmptyLane-Sensor	38	Sensor for checking if the assembly line is empty

A.3 List of Software components

This section gives an overview of the Function Blocks, Programs, and Tasks used in the example. The original software contains more than 370 Function Blocks, hierarchically structured and contained in round about 35 entry level Function Blocks. The root elements for the *Unit1*, *Unit2*, and *Transport* components are specified as Programs. Each Program contains one or more Function Blocks. Table A.3 list the Function Blocks that are used to model the software of the automation system. In contrast to the list of system and environment elements, contains this list only types and not instances created by the Programs or other Function Blocks.

Table A.2: List of Function Blocks

id	Name	Description
1	ShuttleControl	Function Block to control a shuttle. This in-cludes the movement as well as the fixing of the workpiece on the shuttle.
2	Measurment-StationCtrl	This block is used to control all aspects of the measurement. This includes the actua-tors to fix the workpiece, the sensors to me-asure the quality of the workpiece, cleanup the data and providing the data to external sources.
3	Engravement-StationCtrl	The engravement process is controlled by this Function Block. Also the actuators to fix the workpiece on the station are controlled by this block.
4	Rejects-SlideCtrl	A simple Function Block to abstract the sen-sors of the rejects slide.
5	MillCtrl	The mill is not controlled directly by the PLC via IEC Code, but runs on a CNC core. Ho-wever, the interface to this core to exchange status information an commands is imple-mented in this function block.

Table A.2: List of Function Blocks

id	Name	Description
6	Workpiece-DeliveryCtrl	This Function Block controls all sensors and actuators in the workpiece delivery module. It can be used to check whether workpieces are available and on which lane they reside.
7	Bufferstation-Ctrl	This block manages the buffer station. It contains look-ahead functionality to either prepare the buffer for storage or to provide buffered workpieces for the next processing step.
8	RT-RobotCtrl	Function Block that functions as an interface to the robots own PLC. The interface prio-vides commands that are send to the robot and returns status of each executed process.
9	HFT-RobotCtrl	Function Block that functions as an interface to the robots own PLC. The interface prio-vides commands that are send to the robot and returns status of each executed process.
10	FT-RobotCtrl	Function Block that functions as an interface to the robots own PLC. The interface prio-vides commands that are send to the robot and returns status of each executed process.
11	HMIData	This Function Block is used to update, pre-pare and provide the necessary data that is requested from the HMI. It is not used to ac-tually visualize the data on the HMI.
12	SCADAData	This Function Block is used to update, pre-pare and provide the necessary data that is requested from the SCADA system.
13	SPCStationCtrl	Similar to the reject slide does this block con-trol the sensors and actuators of the spc sta-tion.
14	AssemblyConnectorCtrl	A Function Block to manage the assembly connector. The workpiece must be positio-ned on a carrier. The carrier provides three positions. The sensors and actuators to con-trol the carrier in this station are controlled by this block.

The number of Programs used for a specific automation task is usually solely dependent on the developers and existing automation systems that can be reused. Also, depending on the size, its modularity, and the used PLC and development IDE, the functionality of an automation system is either split up into different Function Blocks and executed by just one program or one program for each individual step in an production process. The Turbocharger example makes use of several Programs that are triggered by tasks. Each Program contains a set of Function Blocks that are composed of further Function Blocks.

Table A.3: List of used Programs

id	**Name**	**Description**
1	MillProgram (U1)	Program to control the milling operation for the Unit1. It uses Function Blocks to control the shuttles.
2	MainProgram (U1)	Main program for the Unit1. Controls the RejectsSlide, RT-Robot, and the Workpiece-Delivery. Provides data to the HMI.
3	TransportProgram (T)	Main program controlling the different modules in the Transport unit. It includes FB to control the SPS-Station, BufferStation, SupplyDistribution and more.
4	MeasurementProgram (T)	The measurement is a standalone program for the Transport unit. It controls the Measurement station (with its FBs) and the Engravementstation.
5	MainProgram (U2)	Main program for the Unit2. Controls the RejectsSlide, RT-Robot, and the Assembly-Connector. Provides data to the HMI.
6	MillProgram (U2)	Program to control the milling operation for the Unit2 and its surrounding shuttles.
7	MeasurementProgram (T)	Program to control the Measurementstation and the Engravementstation.